DIF - A Language for Dataflow Graph Specification and Exchange

October 27, 2004

Sponsored by

Defense Advanced Research Projects Agency (DOD)
(Controlling DARPA Office)

ARPA Order C043/70

Issued by U.S. Army Aviation and Missile Command Under

DAAH01-03-C-R236

Management Communications and Control, Inc. (MCCI)
Christopher. B. Robbins
2719 N. Pollard St
Arlington, VA 22207

University of Maryland
Shuvra.S. Bhattacharyya
Office of Research Admin
Lee Building
College Park, MD 20742-5141

Effective Date: August 27, 2003
Short Title: DIF

Contract Expiration Date: August 31, 2004
Reporting Period: August 27, 2003 - August 31, 2004

Section 1 Executive Summary

1.0 Introduction

A major impediment to broad acceptance of dataflow specifications for high performance applications is the absence of any standard for dataflow language. While virtually all dataflow graph based specification methods and supporting software tools are based on a common dataflow mathematical model, they are never the less all mutually incompatible. There are a number of dataflow tools and environments that either execute the dataflow specifications directly or automatically translate the specifications into source code for executable realizations of the specifications. Transferring dataflow graphs or reusing legacy designs and specifications from a different dataflow programming environment is not possible. In the absence of anything like ANSI language standards, developers are reluctant to utilize the promising dataflow technology because using a particular vendors tools is a log term commitment to the vendor as well as the technology. The fact that dataflow tool vendors are for the most part small businesses only exacerbates the problem.

The Dataflow Interchange Format (DIF) is an ongoing research project at the University of Maryland (UMD). Its objective is the development of a vendor neutral language for specification of dataflow graphs. Its acceptance as an industry standard by dataflow tool vendors would enable exchange of dataflow graphs among supporting vendors tools and programming environments. Developers could save and reuse legacy dataflow graphs from all vendors tools used in past developments. Government could provide dataflow graph specifications in DIF from other contracts as GFI with the expectation that they could be readily imported into whatever dataflow environment in use.

DIF's application to high performance computing was originally explored under DARPA's PCA program. Management Communications and Control, Inc. (MCCI), then performing under DARPA SBIR SB011-008, developed a DIF generator tool for the Autocoding Toolset® that generated DIF representations of graphs from the PGM textual form, Signal Processing Graph Notation (SPGN). The feasibility of generating DIF representations of dataflow graphs developed in the PGM development environment using Management Communications and Control, Inc. (MCCI)'s proprietary tools was demonstrated.

STTR 03-003 is a collaboration between MCCI and UMD to demonstrate the feasibility of developing DIF as a vendor neutral dataflow specification language that will support the specification of high performance applications and exchange of those specifications among vendor proprietary tools and programming environments. The objective is to provide a candidate standard dataflow language supporting both graph exchange and accumulation of universally reusable dataflow legacy codes.

1.1 STTR Phase I Objectives

The MCCI, UMD team undertook STTR 03-003 with two main objectives. These are:

a. Development of a feasibility version of DIF, and

b. Demonstration of graph exchange between MCCI's Autocoding Toolset® and UC Berkeley's Ptolemy programming environments using DIF.

Expansion of the DIF prototype demonstrated in the earlier PCA support work into a Turing complete feasibility language version was planned. The feasibility version would demonstrate the capability for expressing high performance applications. For acceptance as an industry standard, it is necessary that DIF have the capability to express the complex and stressful applications found in high performance embedded computing systems and anticipated in HPCS program for the next generation of super computing. Of course, this must be completely vendor neutral. The feasibility version of DIF must be supported with export and import tools that enable its generation from and import into specific vendor or academic programming environments.

The "proof of the pudding" is the exchange of dataflow graphs using DIF between two dataflow programming environments that have nothing in common other that the mathematical model for their specifications. We intended to demonstrate the export of an application specified in PGM in the DIF format and implement the exported DIF specification using Ptolemy. PGM is the dataflow graph based methodology developed by the Navy circa 1980 supporting the Karp and Miller dataflow mathematical model. Ptolemy is the specification and simulation environment developed at UC Berkeley with the origins of its dataflow Modeling capabilities in the same seminal Karp and Miller dataflow publication.

The import process was then to be reversed, importing a DIF specification of an application developed with Ptolemy, implementing it in the PGM based Autocoding Toolset®. This round trip was intended to both demonstrate the feasibility of dataflow graph exchange using DIF and uncover requirements for development of a full performance version in phase II of the STTR.

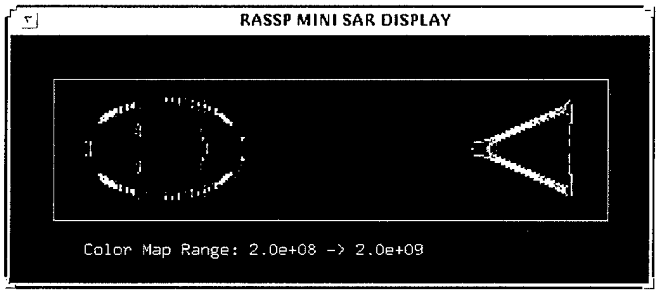
1.2 STTR Phase II Accomplishments

The UMD investigators developed version 0.2 of DIF, incorporating a number of improvements and extensions to version 0.1 developed in the earlier program. The DIF language parser and translation tools for generating Ptolemy applications from DIF specifications were upgraded for the new version of Ptolemy. MCCI upgraded the DIF generator tools to enable generation of DIF version 0.2 from PGM partition graph specifications. A DIF import tool was developed that generated PGM representations of imported DIF application specifications.

The new version of DIF was demonstrated in exchanging a version of the Synthetic Aperture Radar (SAR) benchmark between a realization in MCCI's Autocoding Toolset® and a Ptolemy realization via export and import of its DIF specification. The SAR benchmark has become a standard application for measuring high performance embedded computers and programming environments. It is a stressful application demanding in both processing and inter-processor communications. The round trip was completed by generating a DIF version of the SAR benchmark, re-importing it back into the Autocoding Toolset® programming environment, and generating an application from the imported specification. Both imported versions executed on Ptolemy and cluster architecture targets supported by MCCI with results differing only by precision errors. A second Ptolemy application, Multi Rate Filter, was exported from Ptolemy via DIF and imported into the Autocoding Toolset® programming environment. Figure 1 shows the results of SAR executing in Ptolemy and Autocoding Toolset® environments. This figure represents the phase I bottom line, the round trip exchange of a high performance application using DIF generated in different programming environments for dataflow graph export and import.

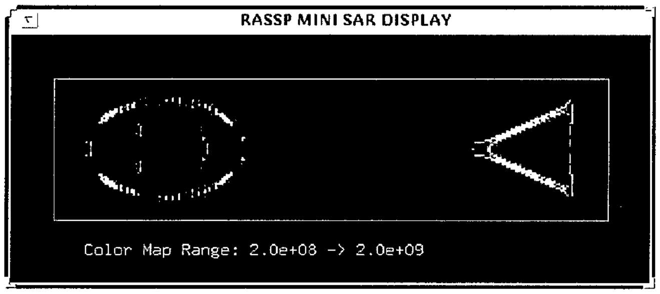
Ptolemy(PGM Export)

```
1.113328370318E9, -5.672582199684E8
1.686243152456E9, -1.132239286739E9
2.280892492213E9, -1.837179778052E9
2.787030647091E9, -2.565079199379E9
3.121469726315E9, -3.124321013999E9
3.235633491442E9, -3.339997173742E9
3.126105298721E9, -3.132702116709E9
2.795907223687E9, -2.578937710771E9
2.292518065694E9, -1.852489499236E9
1.698661416987E9, -1.145532955647E9
```



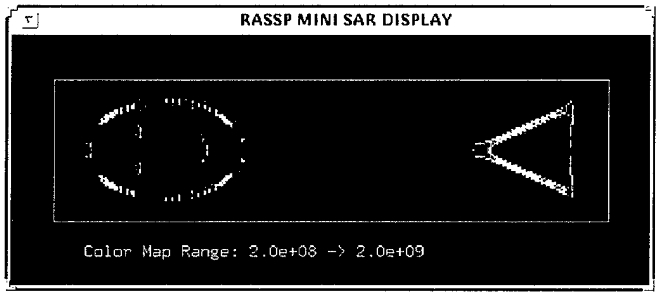
MCCI(DIF Import)

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```



Ptolemy(PGM Export)

```
1.113328370318E9, -5.672582199684E8
1.686243152456E9, -1.132239286739E9
2.280892492213E9, -1.837179778052E9
2.787030647091E9, -2.565079199379E9
3.121469726315E9, -3.124321013999E9
3.235633491442E9, -3.339997173742E9
3.126105298721E9, -3.132702116709E9
2.795907223687E9, -2.578937710771E9
2.292518065694E9, -1.852489499236E9
1.698661416987E9, -1.145532955647E9
```



MCCI(DIF Import)

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```

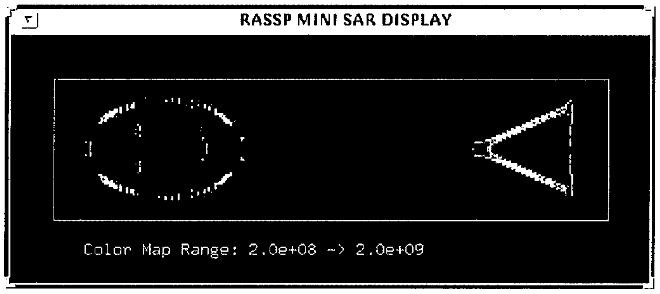


Figure 1 - SAR benchmark application output executing on Ptolemy and on a MCCI's Autocoding Toolset® realization on a Line cluster. DIF was used to export a PGM specification to Ptolemy and a Ptolemy specification of the same application to the Autocoding Toolset® .

The DIF version 0.2 language specification and the two DIF application specifications were an analyzed to determine new capabilities required for a full performance version of DIF. In general the feasibility version of DIF is Turing complete and capable of specifying the class of applications encountered in high performance computing. However it lacks a number of engineering convenience features for handling large and complex applications in a compact manner. MCCI identified requirements for features for an "Industrial Strength" full performance of DIF. UMD and MCCI verified that concepts underlying DIF will support expansion to meet these requirements.

1.3 Study Conclusions

DIF version 0.2 significantly improved the prototype DIF version 0.1. It provides a Turing complete method for specifying dataflow applications. It proved adequate for specification of the defacto standard high performance embedded computing SAR benchmark. Dataflow has often been a difficult concept for object oriented computer scientists to accept. The structure of DIF as a Java Package should make dataflow specification more understandable to a broader range of software engineers and more compatible with existing development environments.

The feasibility of using DIF as a vendor (and university) independent language for exchange of dataflow graphs among proprietary and unique university dataflow environments was demonstrated. The assumption that all dataflow graphs must share a common mathematical model that may serve as a basis for specification exchange was reinforced. Seeming inconsistencies between Ptolemy and PGM representations of dataflow graphs were quickly overcome when analyzed from the viewpoint of dataflow behavior of the underlying model. Constructs with common behavior were quickly identified for topology substitutions. The exchange of the SAR benchmark dataflow graph specifications demonstrated the feasibility of using DIF for graph exchange. Issues identified in the porting demonstration are readily addressable in phase II of the project.

DIF export and import tool prototypes were developed to support graph exchange. Extending these prototypes to support a full performance version of DIF is straightforward involving significant reuse of existing code modules and should pose no development problems.

1.3.1 DIF issues

Several issues were uncovered during the course of the STTR phase I project. These are

a. Requirements for upgrading DIF to provide engineering convenience language features for compact specification of large complex applications targeting parallel architectures.

b. The need for a standard functional library for actors/primitives/ node functional specifications.

While Turing completeness was demonstrated in the development of DIF V0.2, a number of features and capabilities were identified that will facilitate specification of large complex applications. While not strictly necessary, their inclusion in the language definition can significantly improve the language power enabling succinct specifications of complex topology and control interfaces. While DIF is textual, its information must support graphical displays where application dataflow must be intuitive. Verbosity leading to confusing graphical displays is unhelpful. More sophisticated hierarchical structuring, the use of icons representing graph entity arrays, and the use of expressions in control interfaces are important capabilities to add to DIF to enhance its power to specify high performance applications succinctly. Other desired features of secondary importance were also identified.

The problem of non standard functional libraries in dataflow tools was encountered in the export and import experiments. DIF is actor independent, naming actors but not specifying their functionality. Nevertheless when implementing DIF specifications in a particular programming environment, actors or Domain Primitives, native to the target environment must be specified for the application to be realized as executable code. DIF specifications of the SAR benchmark were exported naming Domain Primitives. The specifications had to be transformed to Ptolemy Actor specifications to import the application into Ptolemy. Actor to Domain Primitive transformation was required for importing Ptolemy applications into the Autocoding Toolset®. An Actor Interchange Format tool prototype that facilitates transformation was demonstrated but this represents only a partial solution to the problem of transforming functional specifications within the DIF application graph between all possible combinations of source and target environments. The need for a common functional library as part of the DIF standard became clear during the feasibility study.

1.4 Phase II Objectives

The study identified three main objectives for a phase II follow-on to the phase I feasibility study . These are:

a. Development of DIF V1.0 as an "industrial strength", publicly releasable version of DIF. DIF Version 1.0 will incorporate language features identified in the phase I study enhance its power to specify large, complex high performance applications.

b. Development of a standard functional domain actor library supporting DIF. The VSIPL library will serve as the functional specifications for actor specifications in DIF. Importing tools and environments may implement support for the VSIPL base library or, using the Actor Interchange Format transform DIF VSIPL actor specifications into functionally comparable specifications in the target environment. The VSIPL standard is intended to facilitate exchange of codes between different processing hardware. The addition of a VSIPL based library to DIF will add the capability to exchange specifications between dataflow programming environments.

c. Development of improved infrastructure support for DIF to included full performance versions of DIF export and import tools and code generation frameworks.

1.5 Report Organization

This technical report is organized into 5 sections.. The report sections are:

a. Section 1 - Executive Summary

b. Section 2 - DIF Technical Description. This is a technical overview of version 0.2 of the DIF language and tool support developed for exporting and importing DIF specifications to programming environments

c. Section 3 - Dataflow Graph Exchange This section describes porting an application from MCCI's Autocoding Toolset® dataflow programming environment into the Berkeley Ptolemy environment and then exporting it and re-importing it into MCCI's tools. Exporting a second Ptolemy application is also described. This section develops the supports the study's conclusion that graph exchange via a vendor neutral dataflow language is feasible.

d. Section 4 - Requirements for a full performance DIF. Analysis of DIF version 0.2 capabilities is reported in this section. Capabilities needed for a full performance version are identified and rationales for their requirements are presented. This section provides the foundation for our phase II proposal.

e. Section 5 - Formal Study Conclusions This section states the studies findings and conclusions.

Four appendices are attached to this report. Appendix A is a full description of DIF and the formal language specification. Appendix B includes all listings of all SPGN and DIF specifications generated during the study. Excerpts from these listings are used in the reports main body to illustrate graph import and export. Appendix C is the SableCC grammar. Appendix D is the Actor Interchange Format grammar.

Section 2 DIF Technical Description

2.1 DIF Overview

The Dataflow Interchange Format (DIF) is a standard language to specify mixed-grain dataflow models for digital signal, image, and video processing (DSP) systems and other streaming-related application domains. Major objectives of the DIF project are to design such a standard language; to provide an extensible repository for representing, experimenting with, and developing dataflow models and techniques; and to facilitate technology transfer of applications across DSP design tools.

This phase of the DIF project, in which we discuss in this report, has focused on improving the DIF language and the DIF package to represent more sophisticated dataflow semantics and exploring the capability of DIF in transferring DSP applications and technology.

2.1.1 The DIF Language

The Dataflow Interchange Format (DIF) is proposed to be a standard language for specifying dataflow semantics in all dataflow models. This language is able to be an interchange format for different dataflow-based DSP design tools because it provides an integrated set of syntactic features that can fully capture essential modeling information of DSP applications without over-specification.

From the dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related information. This dataflow semantics specification is based on dataflow modeling theory and independent of any design tool. Therefore, as long as design tools operate based on the supported dataflow models, the DIF language is applicable and the dataflow semantics of a DSP application is unique in DIF regardless of design tools. Moreover, DIF also provides syntax to specify design-tool-specific actor information. Although actor information is irrelevant to dataflow-based analyses, it is essential in exporting, importing, and transferring across tools.

DSP applications specified by the DIF language are usually referred to as *DIF specifications*. The DIF package provides a frond-end tool, the DIF language parser, which converts a DIF specification into a graph-theoretic intermediate representation. This parser is implemented using a Java-based compiler-compiler called SableCC. The complete SableCC grammar of the Dataflow Interchange Format is presented in Appendix B.18.

From DIF language version 0.1 to version 0.2, syntax consistency and code reusability has been improved significantly. DIF language version 0.2 also supports more flexible parameter assignment and provides more flexible syntax in specification attributes. Moreover, it supports most commonly used value types in DSP applications and provides arbitrary naming spaces. Also, the actor block is newly created in DIF version 0.2 for specifying design-tool-dependent actor information.

2.1.2 DIF Language Version 0.2 Syntax

A specification in DIF version 0.2 consists of eight blocks: basedon, topology, interface, parameter, refinement, built-in attribute, user-defined attribute, and actor. Those blocks specify different aspects of dataflow semantics and modeling information. Figure 2.a introduces the syntax of the DIF language version 0.2.

```
modelKeyword graphID {
        basedon { graphID; }
        topology {
                nodes = nodeID, ..., nodeID;
                edges = edgeID (sourceNodeID, sinkNodeID),
                         ...,
                        edgeID (sourceNodeID, sinkNodeID);
        }
        interface {
                inputs = portID [:nodeID], ..., portID [:nodeID];
                outputs = portID [:nodeID], ..., portID [:nodeID];
        }
        parameter {
                paramID;
                paramID = value;
                paramID : range;
        }
        refinement {
                subgraphID = supernodeID;
                subPortID : edgeID; ...; subPortID : portID;
                subParamID = paramID; ...;
        }
        built-in-attribute {
                [elementID] = value;
                [elementID] = id;
                [elementID] = id1, id2, ..., idN;
        }
        attribute user-defined-attribute{
                [elementID] = value;
                [elementID] = id;
                [elementID] = id1, id2, ..., idN;
        }
        actor nodeID {
                computation = stringValue;
                actorAttributeID [: actorAttributeType] = value;
                actorAttributeID [: actorAttributeType] = id;
                actorAttributeID [: actorAttributeType] = id1, ..., idN;
        }
}
```

Figure 2a - Structure of the DIF language version 0.2 syntax

The basedon block provides a useful feature for code reuse. As long as the graph referred to in this has the same topology, interface, and refinement blocks, designers can simply refer to it and override the name, parameters and attributes as desired to construct a new "version" of the same graph structure.

The topology block defines a set of nodes and a set of edges. An edge is specified by its source node identifier and sink node identifier.

The interface block defines a set of input ports and a set of output ports. An interface port can optionally associate inside with a node in the same graph. The inside association is defined in the interface block. An interface port can also connect outside to an edge or a port in the super-graph. This outside connection is defined in the refinement block of the super-graph.

DIF supports parameterization of values in the parameter block. A parameter can be used to parameterize a value, specify a value range. Parameter values can also be left unspecified (e.g., in the case of parameters settings that are dynamically configured). The range of a parameter is specified as a single interval of numbers or a union of multiple intervals.

For each supernode in the graph, there is a refinement block used to specify the supernode to sub-graph refinement. In addition, it specifies the interface connection between the port in the sub-graph and the edge (or port) in the current graph. It also specifies the unspecified sub-graph parameter by the parameter defined in the current graph.

In DIF semantics, nodes, edges, ports, and the graph itself can be associated with attributes. Attributes can be assigned a variety of value types, an identifier (node, edge, port or parameter identifier), or a list of identifiers. Every dataflow model in DIF can define its own built-in attributes and its own methods to process those built-in attributes. The DIF language parser treats built-in attributes in a special way such that the method defined in the corresponding dataflow parser is invoked to handle those built-in attributes. The common built-in attributes defined in DIF are *production*, *consumption*, and *delay*. The *production* and *consumption* attributes specify the token production and consumption rates associated with the corresponding edge, and the *delay* attribute specifies the number of delays associated with an edge.

The user-defined attribute block starts with the keyword *attribute*. Users can define their own attributes, but DIF only records user-defined attributes in special data structures called attribute containers. As far as the DIF language is concerned, all processing of built-in attributes must be implemented by the user, and the DIF package provides convenient interfaces for developing such implementations. The DIF package may also contain functionality that processes built-in attributes. This would be the case for more experimental or specialized functionality that is presently not considered stable or "standard" enough to be incorporated into the DIF language in any form.

The actor block is used to specify design-tool-dependent actor information. Such actor information is irrelevant to platform-independent dataflow-based analyses, such as detection of deadlock or sample-rate inconsistencies in static dataflow models. However, capturing actor information is essential in exporting and importing between DIF and design tools as well as porting DSP applications, because the functionalities of actors need to be preserved. The built-in actor attribute *computation* is used to specify the functionality of a node that represents a functional module (actor) in design tools. DIF provides three built-in actor attribute types, PARAMETER, INPUT, and OUTPUT and supports assigning values, identifiers, and lists of identifiers to actor attributes.

DIF v0.2 provides most commonly used value types in DSP: integer, double, complex, integer matrix, double matrix, complex matrix, string, Boolean, and arrays of the above values. DIF v0.2 also supports scientific notation for double values.

For a complete description of the DIF language version 0.2, please refer to Section 2 and Section 3 in the "Dataflow Interchange Format Version 0.2" technical report.

2.1.4 Dataflow Models

The DIF language is designed to specify all dataflow models for DSP and streaming related applications. In other words, its syntax should be capable of describing dataflow semantics in all dataflow models of computation. DIF version 0.1 has demonstrated its capability of describing CSDF, SDF, single rate dataflow, and HSDF. DIF version 0.2 improves the syntax to support more complicated dataflow semantics, for example, Turing-complete dataflow such as BDF and meta-modeling techniques such as parameterized dataflow. The currently supported dataflow models are CSDF, SDF, single rate graph, HSDF, PSDF, BDF, BCSDF, and ILDF. In addition a new dataflow model of computation, called DIF, is introduced to provide maximum generality when all other supported dataflow models do not match a given application. The DIF model of dataflow imposes minimal restrictions on the characteristics of a dataflow specification; for example, arbitrary objects can be associated with the production rates, consumption rates, and delays of graph edges.

For an introduction to the supported dataflow models in DIF and how to use DIF to specify them, please refer to Section 4 in "Dataflow Interchange Format Version 0.2" technical report.

2.1.3 The DIF package

The DIF package is a Java-based software package developed along with the DIF language. In general, it consists of three major parts: the DIF front-end, the DIF representation, and the dataflow-based analysis, scheduling, and optimization algorithms.

2.1.3.1 The DIF Representation

For each supported dataflow model, the DIF package provides an extensible set of data structures (object-oriented Java classes) for representing and manipulating dataflow graphs in the model. This graph-theoretic intermediate representation for the dataflow model is usually referred to as the DIF representation.

The DIFGraph, corresponding to the DIF model of dataflow described above, is the most general graph class in the DIF package. It represents the basic (minimally-restricted) dataflow graph structure among all dataflow models, and provides methods that are common to all models for manipulating graphs. More specialized dataflow models can be easily incorporated into the DIF representation suite by extending the most closely-related existing graph class, and overriding and adding methods as appropriate to perform more specialized analyses and optimizations. For example, when the synchronous dataflow (SDF) representation was incorporated into the DIF package, the associated graph class was derived from that of cyclo-static dataflow (CSDF), of which SDF is a special case.

Figure 2.b shows the class hierarchy of graph classes in the DIF package. The DIFGraph class is extended from the Graph class in Ptolemy's ptolemy.graph package, which has been developed primarily by the UMD team in collaboration with U. C. Berkeley's Ptolemy group and provides data structures and methods for implementing analyses on generic graphs. The dataflow models CSDF, SDF, single rate dataflow, and HSDF form a "specialization chain" such that each succeeding dataflow model among this list of four is a restricted version (special case) of the previous one. Accordingly, CSDFGraph, SDFGraph, SingleRateGraph, and HSDFGraph form a linear class hierarchy in the DIF package such that each graph class inherits from graph class associated with the previous model in the chain.

In addition to the aforementioned fundamental graph classes, the DIF package also provides the Turing-complete BDFGraph representation, the PSDFGraph representation for modeling of dataflow graph reconfiguration, and the newly proposed BCSDFGraph (binary cyclo-static dataflow), representation, which has been introduced as part of this Phase I work. Furthermore, a variety of other dataflow models are being explored for inclusion in DIF.

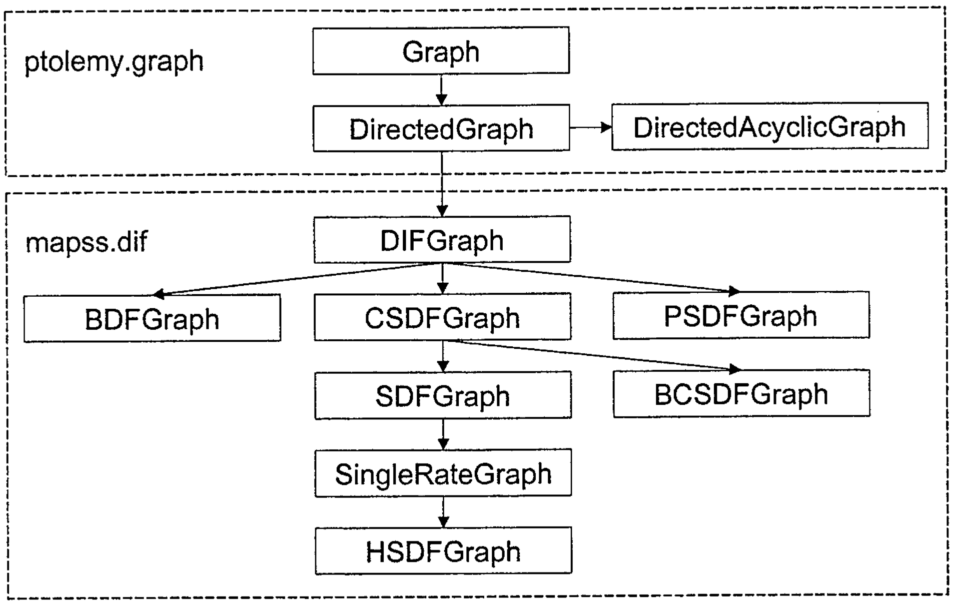


Figure 2.b - The graph class hierarchy in the DIF package

2.1.3.2 The DIF Front-end

Automatic conversion between DIF specifications (.dif files) and DIF representations (graph instances) is perhaps the most fundamental feature of the DIF package. The DIF front-end tool automates this conversion and provides users an integrated set of programming interfaces to construct DIF representations from specifications, and to generate DIF specifications from intermediate representations.

The DIF front-end consists of a Reader class, a set of language parsers (Language Analysis classes), a Writer class, and a set of graph writer classes. The language parsers are implemented using a Java-based compiler compiler called SableCC, which has been developed at McGill University. The flexible structure and efficient Java integration of the SableCC compiler enables easy extensibility for parsing different dataflow graph types.

Figure 2.c illustrates how the DIF front-end constructs the corresponding DIF representation (graph class) from a given DIF specification. The Reader class invokes the corresponding language analysis class (DIF language parser) based on the model keyword specified in the DIF specification. Then, the language analysis class constructs a graph instance according to the dataflow semantics specified in the DIF specification.

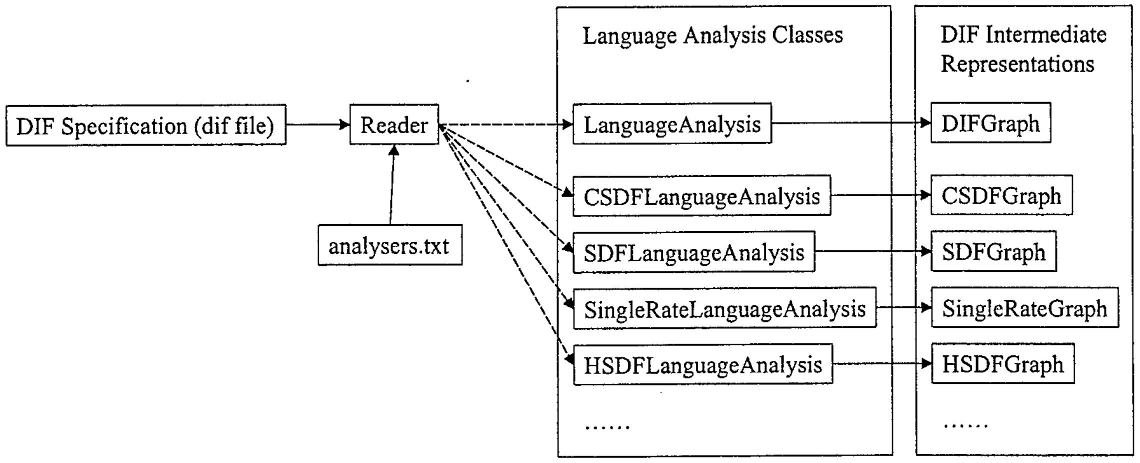


Figure 2.c - The DIF front-end: from DIF specification to DIF representation

On the other hand, Figure 2.d illustrates how the DIF front-end generates the DIF specification according to the DIF representation. The Writer class invokes the corresponding graph writer class based on the type of the given graph instance. After that, the graph writer class generates the DIF specification by tracing elements and attributes of the graph instance.

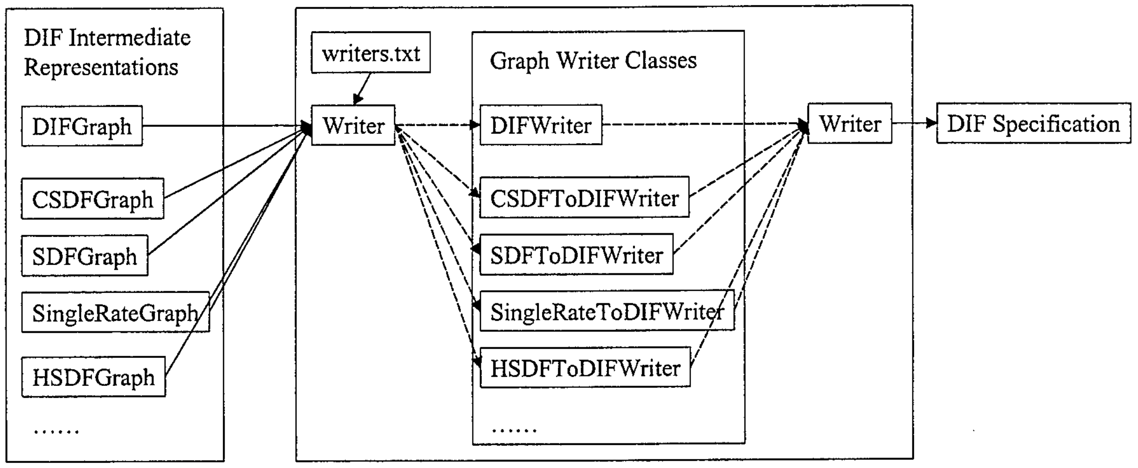


Figure 2.d - The DIF front-end: from DIF representation to DIF specification

2.1.3.3 The Dataflow-based Analysis, Scheduling, and Optimization Algorithms

For supported dataflow models, the DIF package provides not only the graph-theoretic intermediate representation but also various associated analysis, scheduling, and optimization algorithms. Algorithms currently available in the DIF package are based primarily on well-developed algorithms such as iteration period computation, consistency validation, buffer minimization, and loop scheduling. By building on the DIF representations and associated software infrastructure, and invoking the built-in fundamental algorithms as necessary, emerging techniques and newly proposed algorithms can conveniently be designed, implemented, evaluated, and refined using the DIF package.

The dataflow-based algorithms in the DIF package provide designers an efficient interface to analyze and optimize DSP applications. It is also worthwhile to integrate DSP design tools with the DIF package and then utilize the powerful scheduling and optimization features of the DIF package. Indeed, along these lines, we are presently exploring the cooperation of the PSDF domain in Ptolemy II and the PSDF scheduling algorithm in the DIF package. Moreover, integrating SDF related scheduling and optimization techniques in the DIF package with the PGM-based Autocoding Toolset will be explored extensively in the future.

2.1.3.4 Methodology of using DIF

Figure 2.e illustrates the implementation and end user viewpoint of the DIF architecture. DIF supports as the core a layered design methodology covering dataflow models, the DIF language and DIF specifications, the DIF package, dataflow-based DSP design tools, and the underlying hardware and software platforms targeted by these tools. This design methodology clearly demonstrates the careful positioning of DIF in the DSP design and embedded systems communities.

The dataflow models layer represents the dataflow models currently integrated in the DIF package. These models can be further categorized into static dataflow models such as SDF and CSDF; dynamic dataflow models such as the Turing-complete BDF model; and meta-modeling techniques such as parameterized dataflow, which provides the dynamic reconfiguration capability of PSDF. Using the DIF language, application behaviors compatible with these dataflow modeling techniques can be specified as DIF specifications.

The dataflow-based DSP design tools that we have been experimenting with in our development of DIF so far are Ptolemy II developed at UC Berkeley and the Autocoding Toolset developed by MCCI. Tools such as these form a layer in our proposed DIF-based design methodology. Ptolemy II is a java-based design environment and utilizes the Modeling Markup Language (MoML) as its textual format for specification and interchange. Ptolemy II provides multiple models of computation and a large set of libraries consisting of actors for various application domains. On the other hand, the MCCI Autocoding Toolset is based on Processing Graph Method (PGM) semantics and uses Signal Processing Graph Notation (SPGN) as its specification format. It also provides an efficient library consisting of domain primitives for DSP computations and is able to synthesize software implementations for certain high-performance platforms. In our phase I development of DIF, we have been experimenting with our proposed design methodology primarily with the Autocoding Toolset and Ptolemy II.

The hardware / software embedded systems layer gives examples of current hardware / software implementations supported by Ptolemy II and the Autocoding Toolset. Ptolemy II can generate executable Java code for running on the Java VM. On the other hand, the Autocoding toolset is able to generate executable C code for Mercury DSPs and Ada for the Virtual Design Machine (VDM). In addition, we are examining the requirements and implications of DIF-based support for other tools that have the ability to map dataflow models to efficient hardware / software implementations.

The DIF package acts as an intermediate layer between the abstract mathematical properties of dataflow models and implementations of DSP applications on embedded computing platforms. It takes on the responsibility of providing comprehensive data structures for managing dataflow graphs and for carrying out useful dataflow-based algorithms efficiently on these data structures. DIF exporting and importing tools automate the process of exporting DSP applications from design tools to DIF specifications and importing them back to design tools. Automating the exporting and importing processes between DIF and design tools provides the DSP design industry a convenient front-end to use DIF and the DIF package.

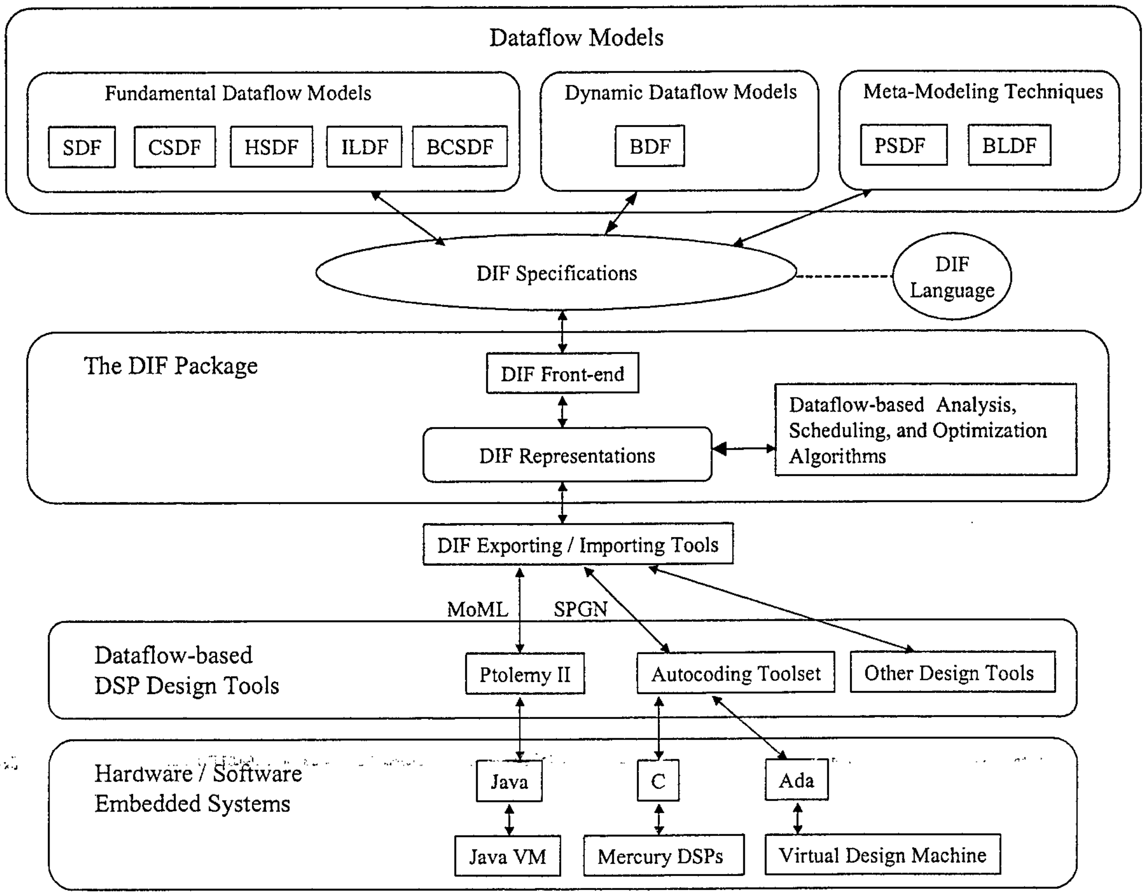


Figure 2.e - Methodology for using DIF.

For a complete report of the DIF project developed in UMD, please refer to "Dataflow Interchange Format Version 0.2" technical report.

2.2 Autocoding Toolset® Tool Support

The Autocoding Toolset® supports DIF with export and import tools. A demonstration DIF export tool was developed in an earlier SBIR, SB011-008. This tool was upgraded to a DIF Generator prototype tool in this effort. This tool translates a SPGN representation of a "flat" graph to a DIF representation. The term "flat", means that the graph contains nodes and queues only. Subgraph hierarchies and families (arrays of entities) and not included in a "flat" graph. "Flattening" refers to the process of expanding subgraph hierarchies and families (arrays of entities) into flat graph topology referencing each entity explicitly.

A DIF import prototype tool was also developed under this SBIR. This tool translates DIF inputs to equivalent PGM graphs in SPGN formats.

2.2.1 DIF Export Tool

Figure 2.f illustrates the organization of the core tools in the Autocoding Toolset®. The Partition Builder tool accepts the input of top level dataflow graphs specifications of applications. These graphs may include subgraph hierarchies, families or arrays of elements, parameters bound at either compilation of instantiation to make specific instances of more general purpose application specifications, and variables changeable during execution to change active graph topology or processing behavior. These application specifications must be partitioned into graph segments that specify the processing of the threads that will ultimately implement the applications processing. Partition graphs are generated for each user specified partition. Partition graphs are flat. Value sets must be enumerated for parameters and runtime variable controls. Most importantly, partition graphs must be cyclo-static, their execution behavior must be periodic. Partitioning rules insure that this latter condition is always met, even if it results in one node partitions.

MPIDGen is the translation and code generation tool. Behavior models are created for each partition graph. The model includes the periodically repeating sequence of the graphs node executions and the memory states of internal queues between each node execution. This behavior model is combined with the Domain Primitive functional specifications for each node and the Target Primitive Maps that provide instructions for implementing the Domain Primitives processing using target math library routines to form a complete specification for the threads code. Thread executable code is then generated from the specification.

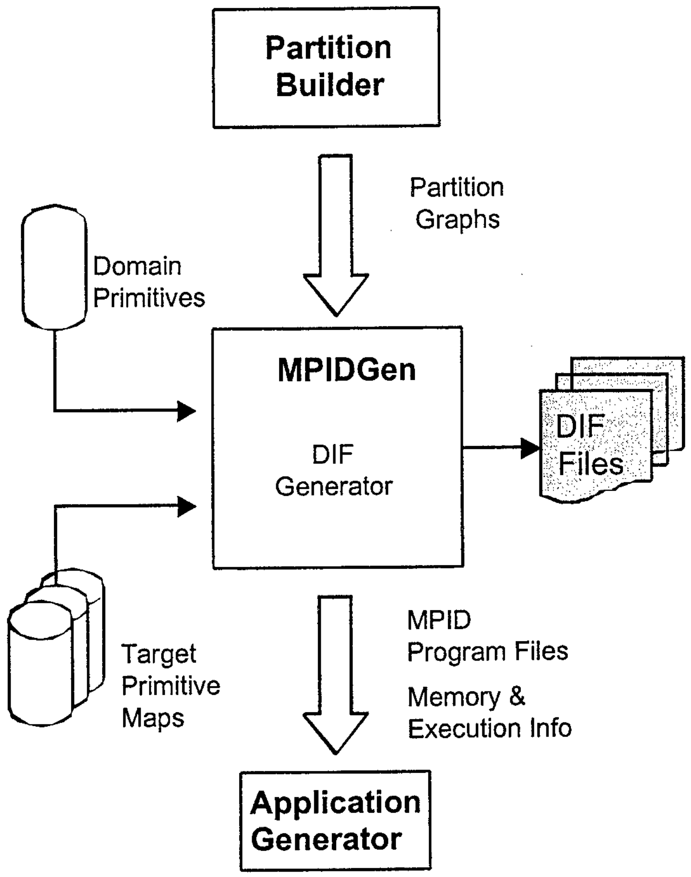


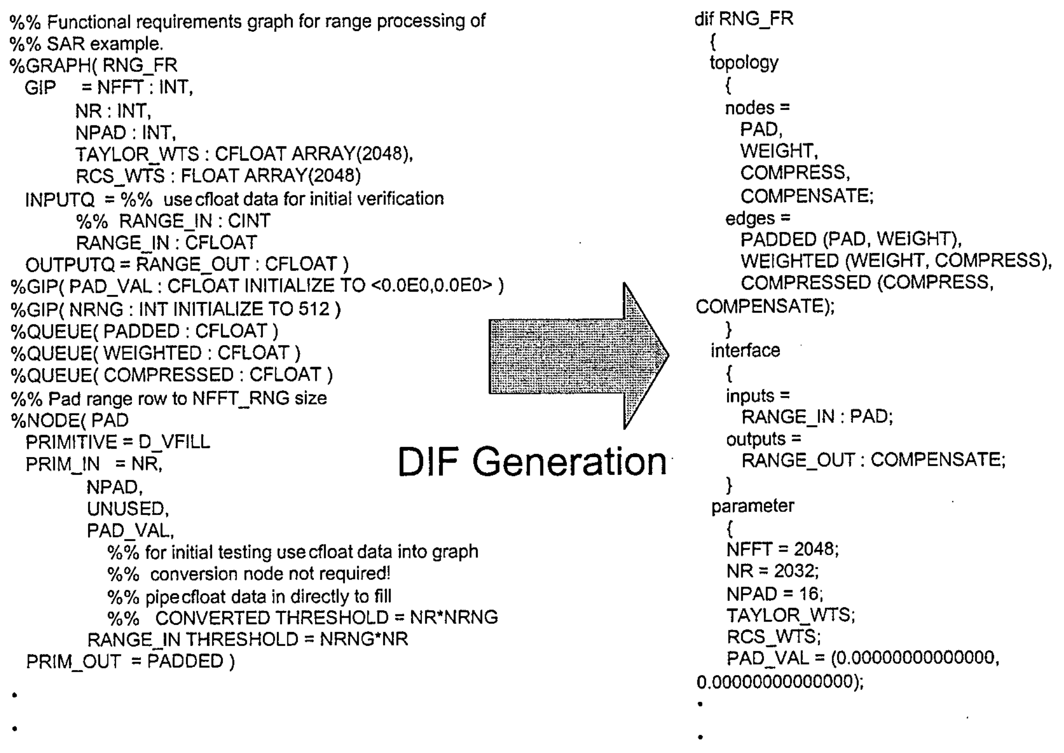
Figure 2.f - MCCI DIF Generator

The Application Generator completes building application source code. Task manager code is generated to manage all tasks assigned to each processing resource. Inter Processor Communication routines are automatically generated to transfer data passed by queues spanning partition specifications. And, a makefile hierarchy is generated enabling compilation with a single make command.

The flat partition graph most nearly matches the current capabilities of DIF. DIF does support hierarchy but not graph entity arrays. Therefor the DIF generator was implemented as a processing path in the MPIDGen tool. SPGN files of partition graphs are created by MPIDGen and the DIF Generator is invoked on each SPGN file producing a translation to DIF. The DIF Generator may be invoked by a command line control. Using the DIF Generator, DIF realizations of partition graphs may be generated for each user specified graph partition.

Figure 2.g is an excerpt from a translation of the range processing in a synthetic radar (SAR) application. The PGM SPGN is shown on the left. The

DIF translation is shown on the right. A complete listing of the two files is contained in Appendix B. MPIDGen may be executed independently on whole graphs. Provided no unsupported graph entities are included, MPIDGen with the DIF generation control set may be used to translate PGM graphs to DIF independent of the other tools. However, going through the partition builder will insure only fully supported flat graphs are translated.



```
%% Functional requirements graph for range processing of
%% SAR example.
%GRAPH( RNG_FR
  GIP    = NFFT : INT,
         NR : INT,
         NPAD : INT,
         TAYLOR_WTS : CFLOAT ARRAY(2048),
         RCS_WTS : FLOAT ARRAY(2048)
  INPUTQ  = %%  use cfloat data for initial verification
         %%  RANGE_IN : CINT
         RANGE_IN : CFLOAT
  OUTPUTQ = RANGE_OUT : CFLOAT )
%GIP( PAD_VAL : CFLOAT INITIALIZE TO <0.0E0,0.0E0> )
%GIP( NRNG : INT INITIALIZE TO 512 )
%QUEUE( PADDED : CFLOAT )
%QUEUE( WEIGHTED : CFLOAT )
%QUEUE( COMPRESSED : CFLOAT )
%% Pad range row to NFFT_RNG size
%NODE( PAD
  PRIMITIVE = D_VFILL
  PRIM_IN  = NR,
          NPAD,
          UNUSED,
          PAD_VAL,
            %% for initial testing use cfloat data into graph
            %%  conversion node not required!
            %% pipe cfloat data in directly to fill
            %%   CONVERTED THRESHOLD = NR*NRNG
          RANGE_IN THRESHOLD = NRNG*NR
  PRIM_OUT  = PADDED )
 .
 .
```

```
dif RNG_FR
  {
  topology
    {
    nodes =
      PAD,
      WEIGHT,
      COMPRESS,
      COMPENSATE;
    edges =
      PADDED (PAD, WEIGHT),
      WEIGHTED (WEIGHT, COMPRESS),
      COMPRESSED (COMPRESS,
COMPENSATE);
    }
  interface
    {
    inputs =
      RANGE_IN : PAD;
    outputs =
      RANGE_OUT : COMPENSATE;
    }
  parameter
    {
    NFFT = 2048;
    NR = 2032;
    NPAD = 16;
    TAYLOR_WTS;
    RCS_WTS;
    PAD_VAL = (0.00000000000000,
0.00000000000000);
 .
 .
```

Figure 2.g - Autocoding Toolset® DIF Generation Example

2.2.2 DIF Import Tool

Figure 2.h illustrates the organization of the DIF import tool implemented under this STTR as a stand alone tool. While it was originally considered the more difficult tool of the pair, it proved possible to leverage UMD DIF Parser and Data Structures and the underlying U.C. Berkeley Ptolemy2 code to simplify the implementation. A DIF to SPGN class was added to the existing modules to transform DIF data structures to SPGN data structures. Once transformed, the existing SPGN generator could generate SPGN from the transformed data structures. The ease of this implementation is strong reinforcement of the commonality of all dataflow entity behavior. The fact that different data structures created from different textual representation of iconically different graphs contain the same information is a clear indication of common underlying mathematical models.

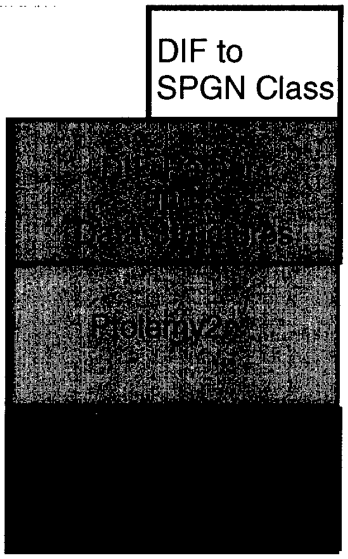


Figure 2.h - Autocoding Toolset® DIF Import Tool

```
dif AZI_FR {
 topology {
  nodes = CORNERTURN, FFT,
CONVL, IFFT;
  edges = YFCO (CORNERTURN,
FFT),
        Y_AZ (FFT, CONVL),
        VMAUL (CONVL, IFFT);
 }
 interface {
  inputs = AZI_IN : CORNERTURN,
         AZ_KERN : CONVL;
  outputs = AZI_OUT : IFFT;
 }
 parameter {
  NFFT;
  RNG_FFT;
 }
 actor CORNERTURN {
  computation = "D_MTRAN";
  M = NFFT;
  N = RNG_FFT;
 •
```

```
%GRAPH (AZI_FR
  GIP =
    NFFT
    RNG_FFT
  INPUTQ =
    AZI_IN
    AZ_KERN
  OUTPUTQ = AZI_OUT
  )
 %QUEUE (YFCO)
%QUEUE (Y_AZ)
%QUEUE (VMAUL)
%NODE (CORNERTURN
  PRIMITIVE =
D_MTRAN
  PRIM_IN =
  PRIM_OUT =
    M = NFFT
    N = RNG_FFT
    X = AZI_IN
    Y = YFCO
•
```

Figure 2.i - Autocoding Toolset® DIF Import Example

Figure 2.i is an example of a translation of a DIF specification of the Azimuth processing graph in the SAR application. DIF is on the left and SPGN is on the right. The tools is a stand alone tool invoked on a DIF input file producing a SPGN file. A full listing of each file is included in appendix B.

on the right. The tools is a stand alone tool invoked on a DIF input file producing a SPGN file. A full listing of each file is included in appendix B.

Section 3 Graph Exchange

3.1 Graph Exchange Overview

The graph exchange experiment conducted by MCCI and UMD was the intended critical technology demonstration in this STTR. The bottom line of the project is that graphs were successfully exchanged between MCCI's Autocoding Toolset® and U.C. Berkeley's Ptolemy programming environments using DIF as the common dataflow language for exchange. The Synthetic Aperture Radar (SAR) benchmark was used as the application for this experiment. The SAR benchmark has become the defacto standard benchmark for embedded high performance computing in that it is relatively simple but very stressful of processing and communications capabilities of target processing systems. Its exchange between dissimilar commercial and academic programming environments and hardware targets using DIF clearly demonstrates DIF's potential to become the standard dataflow graph intended. A second graph, Multi Rate Filter originally developed in Ptolemy, was also imported into the Autocoding Toolset® environment using DIF as a second example of vendor independent graph exchange.

3.2 Exporting the SAR Benchmark

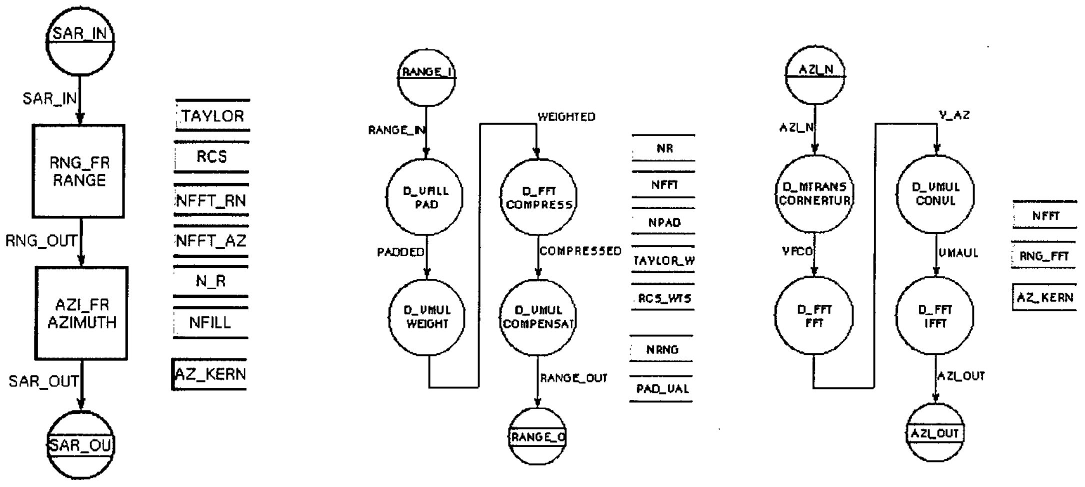


Figure 3.a - Synthetic Aperture Radar Benchmark used in graph exchange experiment

Figure 3.a is the iconic representation of the SAR benchmark. The top level specification included two subgraphs for range and azimuth process. For each received pulse corresponding to a range bin, return data is resolved into frequency or doppler bins. Azimuth processing convolves the range history for each doppler or frequency bin with a range spreading convolution kernel to sharpen range responses. Range/Frequency data matrix is corner turned (transposed) between range and azimuth processes. Range and Azimuth graphs referenced by subgraphs are shown in the figure as well. The range and azimuth processing may be distributed over a number of processors, but in this example both were assigned to a single processing resource. Figure 3.b is an excerpt of the textual representation of the benchmark. A full listing is in Appendix B.

```
%% Functional Requirements Graph for RASSP SAR example One Polarization, full
%% processing; Carl Ecklund, MCCI
%GRAPH( FR_SAR
  GIP     = TAYLOR : CFLOAT ARRAY(2048 ),)
          %% Taylor weighting to reduce sidelobes of compressed pulse.
          RCS : FLOAT ARRAY(2048
          %% Compressed pulse radar cross section weights.
  VAR     = AZ_KERN : CFLOAT ARRAY(1024)
          %% Azimuth convolution kernel - Selection based on slant range.  Total of 31 kernels.
16 used in processing
        %% one frame.  Selection is based on range row being processed.  Same kernel is used
        %% for 128 rows.
         INPUTQ  =  SAR_IN : CINT
          %% Complex integer data that has been FIR filtered
          %% using cfloat data set SAR_IN : CFLOAT
  OUTPUTQ = SAR_OUT : CFLOAT )
          %% Processed data out
%GIP( NFFT_RNG : INT INITIALIZE TO 2048 )
%GIP( NFFT_AZI : INT INITIALIZE TO 1024 )
%GIP( N_R : INT INITIALIZE TO 2032 )
%GIP( NFILL : INT INITIALIZE TO NFFT_RNG-N_R )
%QUEUE( RNG_OUT : CFLOAT INITIALIZE TO (NFFT_RNG*NFFT_AZI)/2 OF
<0.0E0,0.0E0> )
%% Graph RNG_FR implements functional requirements for range processing
%SUBGRAPH( RANGE
  GRAPH   = RNG_FR
  GIP     = NFFT_RNG,
          N_R,
          NFILL,
          TAYLOR,
          RCS
  INPUTQ  = SAR_IN
  OUTPUTQ = RNG_OUT )
•
•
```

Figure 3.b - Textual representation of top level SAR benchmark graph using Signal Processing Graph Notation (SPGN)

3.3 SAR Benchmark Export

The SPGN files for the SAR Range and Azimuth were each processed through the tools as stand alone single partition graphs. The resulting partition graphs are identical to input graphs in that case. The DIF generation control was set to cause output of DIF file translations. Figure 3.c is excerpts of the SPGN input and tool generated DIF output files for the Range processing. Figure 3.d is excerpts of the SPGN and tool generated DIF for the Azimuth processing. Complete listings are shown in Appendix A.

```
%% Functional requirements graph for range processing of
%% SAR example.
%GRAPH( RNG_FR
  GIP    = NFFT : INT,
           NR : INT,
           NPAD : INT,
           TAYLOR_WTS : CFLOAT ARRAY(2048),
           RCS_WTS : FLOAT ARRAY(2048)
  INPUTQ  = %%  use cfloat data for initial verification
           %%  RANGE_IN : CINT
           RANGE_IN : CFLOAT
  OUTPUTQ = RANGE_OUT : CFLOAT )
%GIP( PAD_VAL : CFLOAT INITIALIZE TO <0.0E0,0.0E0> )
%GIP( NRNG : INT INITIALIZE TO 512 )
%QUEUE( PADDED : CFLOAT )
%QUEUE( WEIGHTED : CFLOAT )
%QUEUE( COMPRESSED : CFLOAT )
%% Pad range row to NFFT_RNG size
%NODE( PAD
  PRIMITIVE = D_VFILL
  PRIM_IN   = NR,
            NPAD,
            UNUSED,
            PAD_VAL,
              %% for initial testing use cfloat data into graph
              %%  conversion node not required!
              %% pipe cfloat data in directly to fill
              %%   CONVERTED THRESHOLD = NR*NRNG
            RANGE_IN THRESHOLD = NRNG*NR
  PRIM_OUT  = PADDED )
•
•
```

DIF Generation

```
dif RNG_FR
  {
  topology
    {
    nodes =
      PAD,
      WEIGHT,
      COMPRESS,
      COMPENSATE;
    edges =
      PADDED (PAD, WEIGHT),
      WEIGHTED (WEIGHT, COMPRESS),
      COMPRESSED (COMPRESS,
COMPENSATE);
    }
  interface
    {
    inputs =
      RANGE_IN : PAD;
    outputs =
      RANGE_OUT : COMPENSATE;
    }
  parameter
    {
    NFFT = 2048;
    NR = 2032;
    NPAD = 16;
    TAYLOR_WTS;
    RCS_WTS;
    PAD_VAL = (0.00000000000000,
0.00000000000000);
•
•
```

Figure 3.c - Excerpts of SPGN input and tool generated DIF output for Range processing graph for the SAR benchmark.

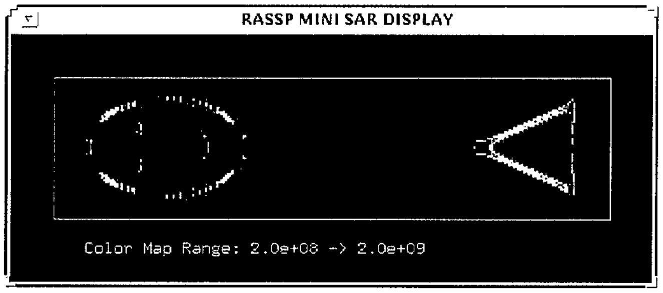
```
%% Functional Requirements for azimuth processing
%%
%GRAPH( AZI_FR
  GIP    = NFFT : INT,
         RNG_FFT : INT
  VAR    = AZ_KERN : CFLOAT ARRAY(1024)
  INPUTQ  = AZI_N : CFLOAT
  OUTPUTQ = AZI_OUT : CFLOAT )

%QUEUE( YFCO : CFLOAT )
%QUEUE( Y_AZ : CFLOAT )
%QUEUE( VMAUL : CFLOAT )

%% Cornerturn the data using a matrix transpose
operation.
%%
%NODE( CORNERTURN
  PRIMITIVE = D_MTRANS
  PRIM_IN  = NFFT,
          RNG_FFT,
          AZI_N THRESHOLD = NFFT*RNG_FFT
              CONSUME = NFFT*RNG_FFT/2
  PRIM_OUT  = YFCO )
%NODE( FFT
  PRIMITIVE = D_FFT
  PRIM_IN  = NFFT,
          UNUSED,
          UNUSED,
          UNUSED,
          UNUSED,
          YFCO THRESHOLD = NFFT*RNG_FFT
  PRIM_OUT  = Y_AZ )
•
•
```

```
dif AZI_FR
 {
 topology
  {
  nodes =
    CORNERTURN,
    FFT,
    CONVL,
    IFFT;
  edges =
    YFCO (CORNERTURN, FFT),
    Y_AZ (FFT, CONVL),
    VMAUL (CONVL, IFFT);
  }
 interface
  {
  inputs =
    AZI_N : CORNERTURN;
  outputs =
    AZI_OUT : IFFT;
  }
 parameter
  {
  NFFT = 1024;
  RNG_FFT = 2048;
  AZ_KERN;
  }
 production
  {
  YFCO = 2097152;
  Y_AZ = 2097152;
  VMAUL = 2097152;
  AZI_OUT = 1048576;
  }
```

Figure 3.d - Excerpts of SPGN input and tool generated DIF output for Azimuth processing graph for the SAR benchmark.

Machine generated DIF files for Range and Azimuth processing with a hand generated DIF file for the top level graph were provided to UMD for import into the Ptolemy environment. A manually programmed version of the top level graph was necessary because Subgraphs are flattened in the Partition Builder tool and do not exist as partition graphs. This limitation is easily correctable in phase II.

```
1.113328370318E9, -5.672582199684E8
1.686243152456E9, -1.132239286739E9
2.280892492213E9, -1.837179778052E9
2.787030647091E9, -2.565079199379E9
3.121469726315E9, -3.124321013999E9
3.235633491442E9, -3.339997173742E9
3.126105298721E9, -3.132702116709E9
2.795907223687E9, -2.578937710771E9
2.292518065694E9, -1.852489499236E9
1.698661416987E9, -1.145532955647E9
```


RASSP MINI SAR DISPLAY

Color Map Range: 2.0e+08 -> 2.0e+09

Figure 3.e - SAR benchmark data generated from Ptolemy execution of imported DIF plotted in common display format.

3.3 Import of machine generated DIF into Ptolemy

3.3.1 Exporting and Importing Issues

Developing the capability of translating across design tool specification formats and the Dataflow Interchange Format is the most challenging aspect of our work on DIF. It is a difficult but important task to automate the exporting and importing processes. First of all, graph topologies and hierarchical structures of DSP applications must be captured in order to completely represent their dataflow semantics. Furthermore, actor computations, parameters, and connections must also be specified to preserve the functional behavior of applications.

Figure 3.f illustrates the exporting and importing mechanism proposed by UMD.

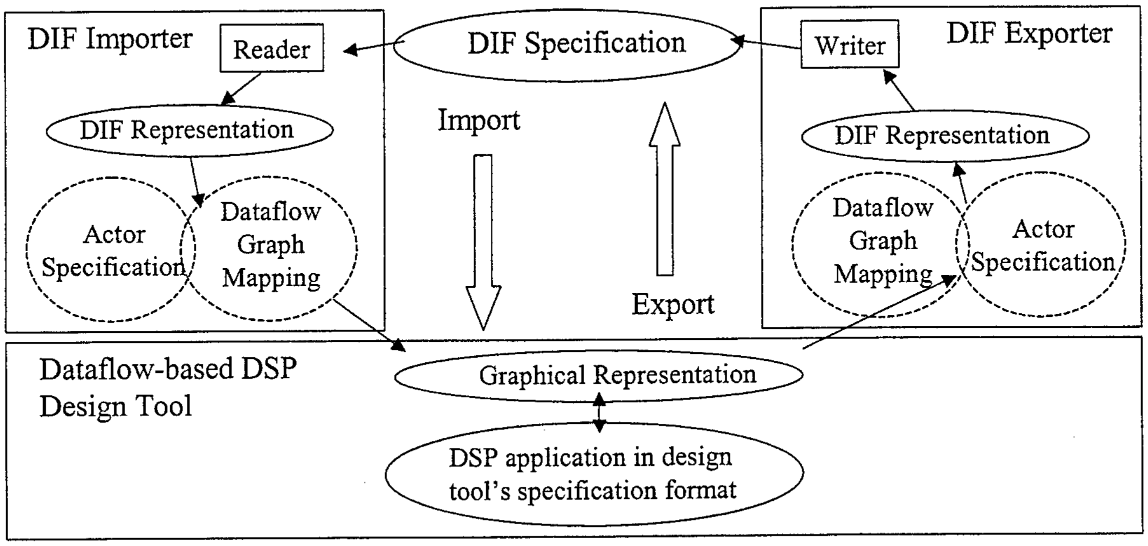


Figure 3.f - Exporting and Importing Mechanism

3.3.2 Dataflow Graph Mapping

Dataflow-based DSP design tools usually have their own representations for nodes, edges, hierarchies, etc. Moreover, they often use more specialized components instead of the formal dataflow representations. Implementation issues in converting the graphical representations of design tools to the formal dataflow representations used in DIF are categorized as *dataflow graph mapping* issues.

For example, Ptolemy II has *AtomicActor* objects for representing DSP computations and *CompositeActor* objects for representing subgraphs. It uses *Relation* objects instead of edges to connect actors. Each actor has multiple *IOPort*s and those *IOPort*s are the connection points for *Relation*s. A *Relation* can have a single source but fork to multiple destinations. A regular *IOPort* can accept only one *Relation* but Ptolemy II also allows *multiport IOPorts* that can accept multiple Relations.

Clearly, special care must be taken when mapping Ptolemy II graphical representations to DIF representations. First, nodes in DIF do not have ports to distinguish interfaces. Second, edges cannot support multiple destinations in contrast to *Relations*. Third, the *multiport* property in Ptolemy II does not correspond exactly to any feature found in generic dataflow representations.

Although such tool-specific implementation problems in dataflow graph mapping arise and are highly tool-specific, through careful use of DIF features and specification configuration mechanisms, exporting without losing any essential modeling information is still feasible. First, the DIF language is capable of describing dataflow semantics regardless of the particular design tool used to enter the semantics as long as the tool is dataflow-based, or supports a modeling style that is sufficiently close to dataflow. Second, DIF representations can fully realize the dataflow graphs specified by the DIF language. Based on these two advantages, our approach to solving the dataflow graph mapping issues between Ptolemy II and DIF is to design an algorithm to traverse graphical representations in Ptolemy II, and then convert the Ptolemy II components encountered during the traversal into equivalent components or groups of components in DIF. After that, our DIF front-end tool can write the DIF representations into textual DIF specifications.

In this algorithm that maps Ptolemy II graphical representations to DIF representations, *AtomicActors* are represented by nodes and *CompositeActors* are represented by hierarchies. Single-source-single-destination *Relations* are represented by edges. For a multiple-destination *Relation*, a fork actor (please refer to Section 6.4 in "Dataflow Interchange Format Version 0.2" technical report) and several edges are used to represent it without losing any dataflow property or functional characteristic. An actor's *IOPorts* and the corresponding connections are specified as actor attributes. Even for a *multiport IOPort*, multiple connections can still be listed in the corresponding DIF attribute.

3.3.3 Actor Specification

Specifying an actor's computation as well as all necessary operational information is referred to as *actor specification*. It is an important issue in exporting and importing between DIF and design tools as well as porting DSP applications across tools because every actor's functionality must be preserved. The actor block is newly incorporated into the DIF language version 0.2 for actor specification. The DIF language syntax for the actor block is described in Section 2.1.2.

To illustrate actor specification, we take the FFT operation in Ptolemy II and MCCI's Autocoding Toolset as examples. In Ptolemy II, actors are implemented in Java and invoked through their class path. The FFT actor in Ptolemy II is defined in the ptolemy.domains.sdf.lib.FFT class of the Ptolemy II SDF library. It has a parameter *order* and two IOPorts, *input* and *output*. Therefore, in the corresponding DIF actor specification, the attribute *order* (with attribute type PARAMETER) specifies the FFT order. In addition, attributes *input* (with attribute type INPUT) and *output* (with attribute type OUTPUT) specify the incomingEdgeID and outgoingEdgeID connecting to the corresponding IOPorts. A full DIF actor block for a Ptolemy FFT actor is presented below:

```
actor nodeID {
        computation = "ptolemy.domains.sdf.lib.FFT;
        order : PARAMETER = integerValue or integerParameterID;
        input : INPUT = incomingEdgeID;
        output : OUTPUT = outgoingEdgeID;
}
```

A FFT domain primitive in Autocoding Toolset, on the other hand, is referred to by D_FFT. In the D_FFT domain primitive, parameter *X* specifies its input, parameter *Y* specifies its output, and parameter *N* specifies its length. The corresponding DIF specification for the D_FFT domain primitive is presented below:

```
actor nodeID {
        computation = "D_FFT";
        N = integerValue or integerParameterID;
        X = incomingEdgeID;
        Y = outgoingEdgeID;
}
```

3.4 Export of DIF specification of SAR Benchmark

The DIF specification of the Autocoding Toolset SAR application is presented in Appendix B. With the actor interchange specification presented in Appendix A.20 and the actor interchange methods developed in the DIF package, the actor mapping mechanism can translate the DIF specification of the Autocoding Toolset SAR application to the DIF specification for Ptolemy II, which is presented in Appendix B.21. Finally, the DIF-to-Ptolemy importer developed by the UMD team imports the DIF specification in Appendix B.21 into Ptolemy II. The ported graphical representation in Ptolemy II is showed in Figure 3.g.

Figure 3.g(a) represents the top-level coarse grain graph of the SAR application in Ptolemy II. The composite actors (block with red borders) RNG_FR and AZI_FR represent the range processing subgraph and the azimuth processing subgraph, respectively. Figure 3.g(b) is the range processing

RNG_FR graph and Figure 3.g(c) is the azimuth processing graph. The node IFFT is mapped to the IFFT_SUBGRAPH in Figure 3.g(d).

The MCCI Autocoding Toolset specifies input/output procedures outside its graph specifications. As a result, I/O actors are added manually in the Ptolemy II design to feed data samples as well as coefficients into the SAR graph and to write and display the results. Figure 3.h shows the SAR application in Ptolemy II after adding I/O actors. The composite actor SAR_FR in Figure 3.h actually represents the top-level SAR in Figure 3.b(a).

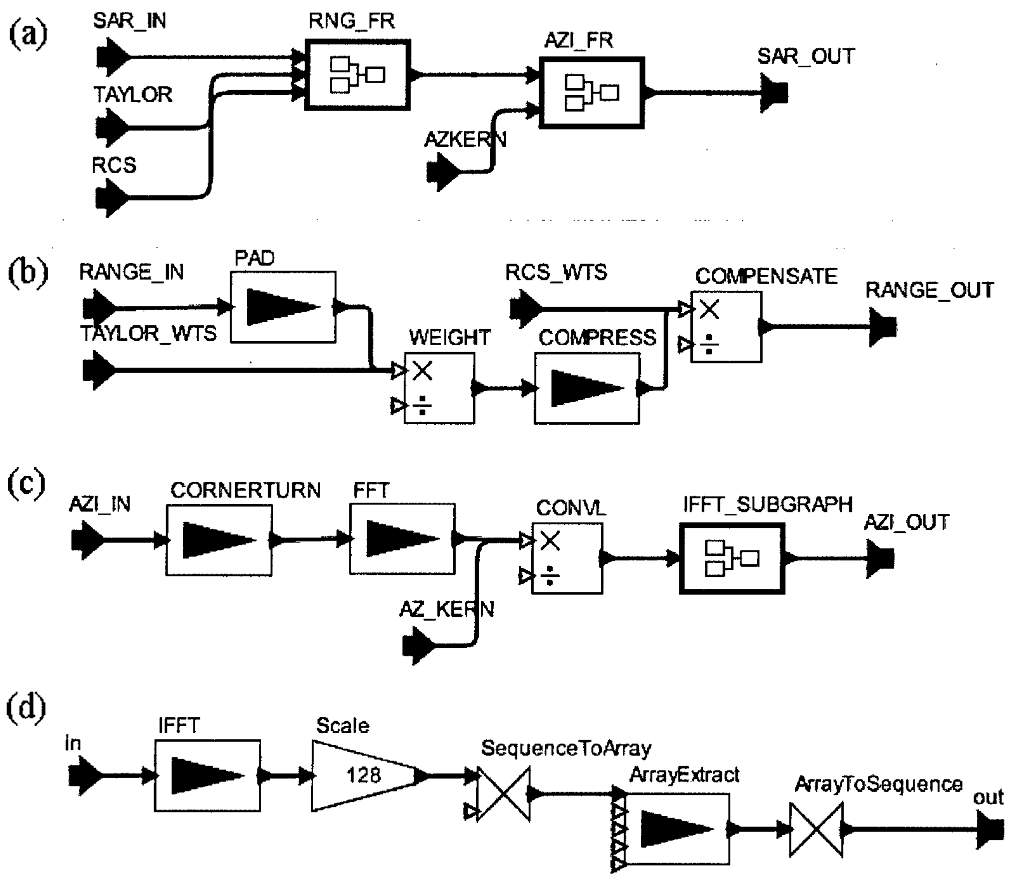


Figure 3.g The SAR benchmark application in Ptolemy.
(a) SAR_FR graph. (b) RNG_FR graph. (c) AZI_FR graph. (d) IFFT_SUBGRAPH graph

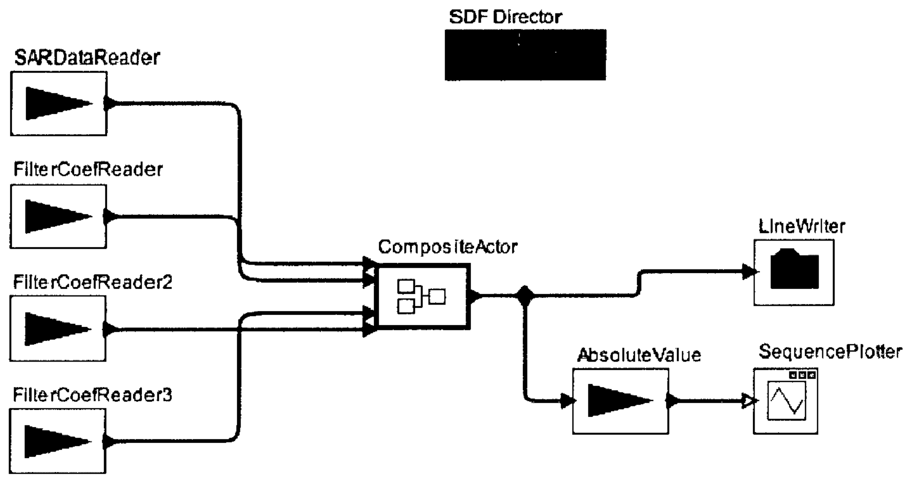


Figure 3.h The SAR benchmark application in Ptolemy after adding I/O actors

3.5. Import of Ptolemy DIF SAR specification into the Autocoding Toolset® programming environment.

The conversion algorithm for translating graphical representations in Ptolemy II to equivalent DIF representations is outlined in Section 3.3.1. The DIF-to-Ptolemy exporter is developed based on this algorithm and is able to export the Ptolemy II graphical representation of the SAR application to a DIF internal representation. Then through the DIF front-end tool, the DIF specification for the Ptolemy II SAR application can be easily generated. This Ptolemy II to DIF exporting process is the reverse of the importing process described in Section 3.4.

DIF specifications of the top level SAR graph and Range and Azimuth subgraphs were generated from Ptolemy implementations by UMD investigators. These graph specifications were translated into SPGN using the DIF import tool. Figure 3.i shows excerpts of the DIF top level graph and the SPGN generated from it by the import tool. Figure 3.j shows excerpts from the Range subgraph import from DIF. Figure 3.k shows excerpts from the Azimuth graph import from DIF. Complete listings of input DIF files and generated SPGN files are contained in Appendix B.

```
dif FR_SAR {
 topology {
  nodes = RANGE, AZIMUTH;
  edges = RNGOUT (RANGE,
AZIMUTH);
 }
 interface {
  inputs = SAR_IN : RANGE,
      TAYLOR : RANGE,
      RCS : RANGE,
      AZKERN : AZIMUTH;
  outputs = SAR_OUT : AZIMUTH;
 }
 parameter {
  NFFT_RNG = 2048;
  NFFT_AZI = 1024;
  N_R = 2032;
  NFILL = 16;
 }
 refinement {
  RNG_FR = RANGE;
  RANGE_IN : SAR_IN;
.
```

SPGN Generation

```
%GRAPH (FR_SAR
  INPUTQ =
    SAR_IN
    TAYLOR
    RCS
    AZKERN
  OUTPUTQ = SAR_OUT
  )
%GIP (NFFT_AZI : INT
  INITIALIZE TO 1024
  )
%GIP (NFFT_RNG : INT
  INITIALIZE TO 2048
  )
%GIP (NFILL : INT
  INITIALIZE TO 16
  )
%GIP (N_R : INT
  INITIALIZE TO 2032
  )
%QUEUE (RNGOUT)
%SUBGRAPH (RANGE
.
```

Figure 3.i - Excerpts of top level DIF specification of the SAR benchmark and the SPGN translated from it by the DIF import tool.

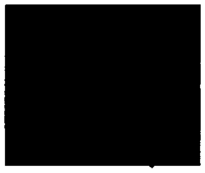
```
dif RNG_FR {
 topology {
  nodes = PAD, WEIGHT, COMPRESS, COMPENSATE;
  edges = PADDED (PAD, WEIGHT),
      WEIGHTED (WEIGHT, COMPRESS),
      COMPRESSED (COMPRESS, COMPENSATE);
 }
 interface {
  inputs = RANGE_IN : PAD,
       TAYLOR_WTS : WEIGHT,
       RCS_WTS : COMPENSATE;
  outputs = RANGE_OUT : COMPENSATE;
 }
 parameter {
  NFFT;
  NR;
  NPAD;
  NRNG = 512;
  PAD_VAL = 0.0;
 }
 actor PAD {
•
•
```

SPGN Generation

```
%GRAPH (RNG_FR
  GIP =
    NFFT
    NPAD
    NR
  INPUTQ =
    RANGE_IN
    TAYLOR_WTS
    RCS_WTS
  OUTPUTQ =
RANGE_OUT
  )
%GIP (NRNG : INT
  INITIALIZE TO 512
  )
%GIP (PAD_VAL : FLOAT
  INITIALIZE TO 0.0
  )
%QUEUE (PADDED)
%QUEUE (WEIGHTED)
•
•
```

Figure 3.j - Excerpts of top level DIF specification of the SAR benchmark Range Subgraph and the SPGN translated from it by the DIF import tool.

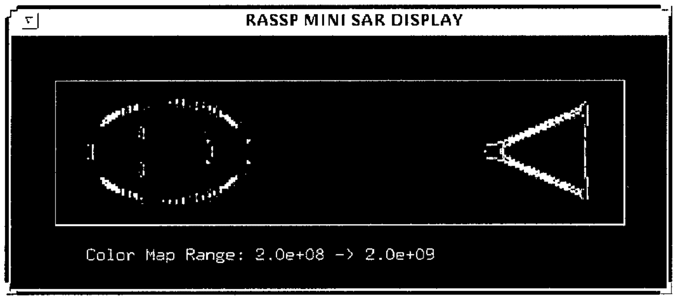
```
dif AZI_FR {
 topology {
  nodes = CORNERTURN, FFT,
CONVL, IFFT;
  edges = YFCO (CORNERTURN,
FFT),
        Y_AZ (FFT, CONVL),
        VMAUL (CONVL, IFFT);
 }
 interface {
  inputs = AZI_IN : CORNERTURN,
         AZ_KERN : CONVL;
  outputs = AZI_OUT : IFFT;
 }
 parameter {
  NFFT;
  RNG_FFT;
 }
 actor CORNERTURN {
  computation = "D_MTRAN";
  M = NFFT;
  N = RNG_FFT;
  •
```



```
%GRAPH (AZI_FR
  GIP =
    NFFT
    RNG_FFT
  INPUTQ =
    AZI_IN
    AZ_KERN
  OUTPUTQ = AZI_OUT
  )
 %QUEUE (YFCO)
%QUEUE (Y_AZ)
%QUEUE (VMAUL)
%NODE (CORNERTURN
  PRIMITIVE =
D_MTRAN
  PRIM_IN =
  PRIM_OUT =
    M = NFFT
    N = RNG_FFT
    X = AZI_IN
    Y = YFCO
•
```

Figure 3.k - Excerpts of top level DIF specification of the SAR benchmark Azimuth Subgraph and the SPGN translated from it by the DIF import tool.

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```



```
Color Map Range: 2.0e+08 -> 2.0e+09
```

Figure 3.l - SAR benchmark output from DIF specification

Figure 3.l shows the output display and a excerpt of the data produced from execution of the version of the SAR application imported from the Ptolemy DIF specification. Comparison of this figure with figure 3.e show that Ptolemy execution of the SAR graph imported from PGM via DIF and the autocoded execution of the benchmark of the version of the benchmark imported from the DIF specification of the Ptolemy implementation differ only in precision errors.

The SAR benchmark has made a round trip from PGM to Ptolemy and from Ptolemy to PGM via DIF.

3.6 Importing Multi-Rate Filter Benchmark from Ptolemy

The multi rate filter bank application is developed using Ptolemy II. It implements an eight-level perfect reconstruction one-dimensional filter bank based on the biorthogonal wavelet decomposition. Figure 3.m shows the graphical representation of this filter bank in Ptolemy II. Through the DIF-to-Ptolemy exporter developed by the UMD team, the DIF specification of this filter bank application is automatically generated. The complete specification is presented in Appendix B.22.

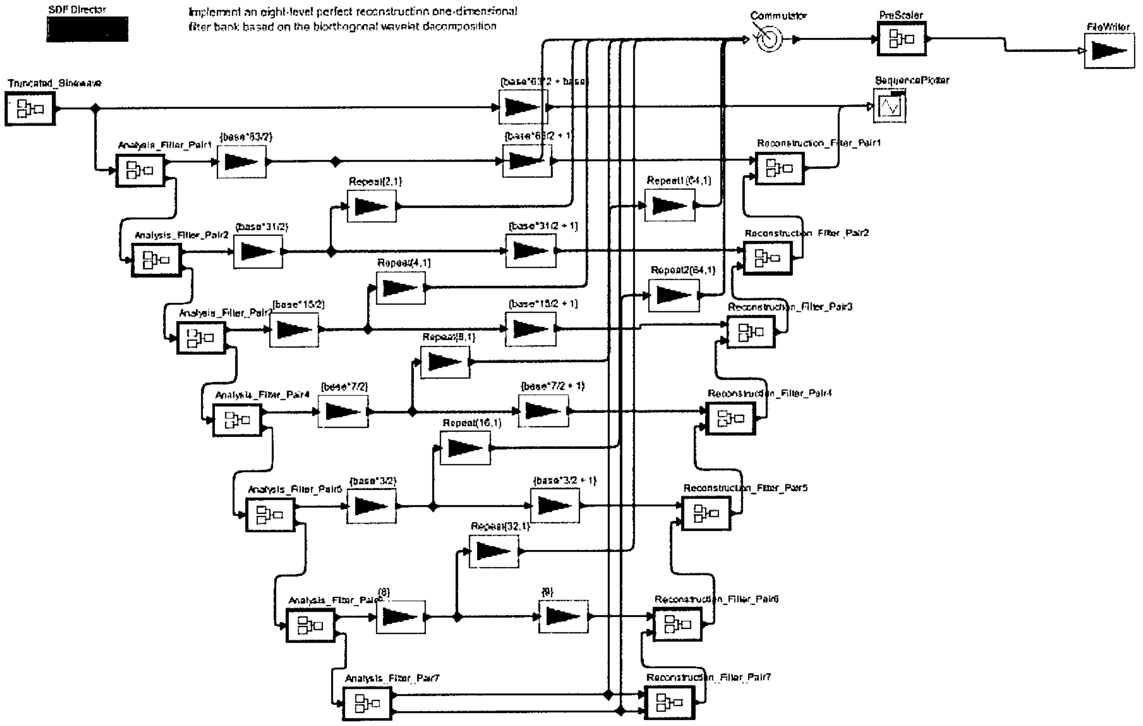


Figure 3.m The filter bank application in Ptolemy II

A second import experiment was conducted as part of this study. The multi-rate filter is a standard Ptolemy benchmark. Figure 3.m shows the Ptolemy iconic representation of this application. DIF was generated for this benchmark. Figure 3.n is an excerpt of the DIF generated from the Ptolemy multi-rate filter benchmark. The full listing is included in appendix B.

```
dif Truncated_Sinewave {
    topology {
        nodes = Ramp,
                Pulse,
                TrigFunction,
                MultiplyDivide,
                ${center/2}$;
        edges = e0 (Ramp, TrigFunction),
                e1 (Pulse, ${center/2}$),
                e2 (TrigFunction, MultiplyDivide),
                e3 (${center/2}$, MultiplyDivide);
    }
    interface {
        outputs = output:MultiplyDivide;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute frequency { = 0.6283185307179586; }
    attribute center { = 50; }
    attribute lengthOfSineBurst { = 50; }
    actor Ramp {
        computation = "ptolemy.actor.lib.Ramp";
        firingCountLimit : PARAMETER = 0;
        init : PARAMETER = -79.57747154594767;
        step : PARAMETER = 0.6283185307179586;
        output : OUTPUT = e0;
    }
    actor Pulse {
        computation = "ptolemy.actor.lib.Pulse";
        firingCountLimit : PARAMETER = 0;
        indexes : PARAMETER =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,5
0};
        values : PARAMETER =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
        repeat : PARAMETER = false;
        output : OUTPUT = e1;
    }
```

Figure 3.n - Excerpt from DIF specification of Ptolemy multi-rate filter benchmark

The DIF version was imported into the Autocoding Toolset® environment by translating the DIF specification to SPGN with the DIF import tool. Figure 3.o shows an excerpt of the SPGN translation of the top level graph. The full listing is included in appendix B.

```
%GRAPH (FilterBank   )

%GIP (_vergilCenter : FLOAT ARRAY(2)
   INITIALIZE TO {536.5,439.5}
   )
%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {-4,12}
   )
□
□
%GIP (noIterations : INT
   INITIALIZE TO 1600
   )

%QUEUE (e35)
□
□
%QUEUE (e96)

%SUBGRAPH (Truncated_Sinewave
   GRAPH = Truncated_Sinewave
   GIP =
   INPUTQ =
   OUTPUTQ =
      output = e35
   )
%SUBGRAPH (Analysis_Filter_Pair1
   GRAPH = Analysis_Filter_Pair1
   GIP =
   INPUTQ =
      input = e36
   OUTPUTQ =
      output1 = e38
      output2 = e39
   )
□
□
```

Figure 3.o - SPGN listing of the top level multi rate filter graph translated from DIF input.

The multi rate filter application includes four subgraphs, Analysis_Filter_Pair, PreScaler, Reconstruction_Filter, and Truncated_Sinewave. Excerpts of the SPGN files translated from these DIF files are shown as figures 3.p through 3.s. Complete listings are included in appendix B.

```
%GRAPH (Analysis_Filter_Pair1
   INPUTQ = input
   OUTPUTQ =
      output1
      output2
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {211,474}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (highpass : <UNKNOWN>
   INITIALIZE TO qmf.highpass.filter
   )
%GIP (lowpass : <UNKNOWN>
   INITIALIZE TO qmf.lowpass.filter
   )

%QUEUE (e4)
%QUEUE (e5)

%NODE (FIR_highpass
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 2
      decimationPhase = 1
      interpolation = 1
      taps = {0.001224,-7.0E-4,-0.011344,0.011408,0.023464,-0.001747,-
0.044403,-0.204294,0.647669,-0.647669,0.204294,0.044403,0.001747,-
0.023464,-0.011408,0.011344,7.0E-4,-0.001224}
      input = e4
      output = output1
   )
%NODE (FIR_lowpass
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 2
      decimationPhase = 1
      interpolation = 1
   taps ={0.001224,-6.98E-4,-0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0226242,-
0.030986,0.071283,0.011682,-0011833,-6.98E-4,0.001224}
•
•
```

Figure 3.p - Excerpt translated from DIF specification of Analysis_Filter_Pair graph

```
%GRAPH (PreScaler
   INPUTQ = input
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {181,466}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (gain : INT
   INITIALIZE TO 100
   )
%GIP (offset : INT
   INITIALIZE TO 128
   )

%QUEUE (e32)
%QUEUE (e33)
%QUEUE (e34)

%NODE (Scale
   PRIMITIVE = ptolemy.actor.lib.Scale
   PRIM_IN =
   PRIM_OUT =
      factor = 100
      scaleOnLeft = true
      input = input
      output = e32
   )
%NODE (Limiter
   PRIMITIVE = ptolemy.actor.lib.Limiter
   PRIM_IN =
   PRIM_OUT =
      bottom = -128.0
      top = 127.0
      input = e32
      output = e33
   )
%NODE (AddSubtract
   PRIMITIVE = ptolemy.actor.lib.AddSubtract
   PRIM_IN =
   PRIM_OUT =
      output = output
   )
•
•
```

Figure 3.q - Excerpt translated from DIF specification of PreScaler graph

```
%GRAPH (Reconstruction_Filter_Pair1
   INPUTQ =
      input1
      input2
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {232,252}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (highpass : <UNKNOWN>
   INITIALIZE TO qmf.highpass.filter
   )
%GIP (lowpass : <UNKNOWN>
   INITIALIZE TO qmf.lowpass.filter
   )

%QUEUE (e18)
%QUEUE (e19)

%NODE (FIR_highpass_R
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 1
      decimationPhase = 0
      interpolation = 2
      taps = {-0.001224,-6.98E-4,0.011833,0.011682,-0.071283,-
0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224}
      input = input1
      output = e18
   )
%NODE (FIR_lowpass_R
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 1
      decimationPhase = 0
      interpolation = 2
      taps = {0.001224,7.0E-4,-0.011344,-0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224}
      input = input2
      output = e19
   )
•
•
```

Figure 3.r - Excerpt translated from DIF specification of Reconstruction_Filter graph

```
%GRAPH (Truncated_Sinewave
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {232,252}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (center : INT
   INITIALIZE TO 50
   )
%GIP (frequency : FLOAT
   INITIALIZE TO 0.6283185307179586
   )
%GIP (lengthOfSineBurst : INT
   INITIALIZE TO 50
   )

%QUEUE (e0)
%QUEUE (e1)
%QUEUE (e2)
%QUEUE (e3)

%NODE (Ramp
   PRIMITIVE = ptolemy.actor.lib.Ramp
   PRIM_IN =
   PRIM_OUT =
      firingCountLimit = 0
      init = -79.57747154594767
      step = 0.6283185307179586
      output = e0
   )
```

-
-

Figure 3.s - Excerpt translated from DIF specification of Truncated_Sinewave graph

While this SPGN produced is perfectly correct, it is incomplete for autocoding. Note that in the SPGN translations the characters <UNKNOWN> appear. The translator inserts this prompt where needed information is missing. Also note that the primitives in all the node statements refer to actors in the Ptolemy library, e.g. ptomemy.actor.lib.Ramp. There are not exact equivalents for all Ptolemy primitives specified in the Domain Primitive Library. Completing the specification to a state where executable code can be generated would involve implementing Domain Primitive equivalents for each Ptolemy primitive referenced. MCCI did not continue this effort since the SPGN produced is in the correct form and implementing Domain Primitives for all uncovered Ptolemy primitives would exceed the phase I scope.

This experiment clearly identified issues that must be addressed in phase II. References to vendor specific primitive library elements in both tools points to the need for a common primitive library. A second need became clear. The functions Analysis _Filter_Pair and Reconstruction_Filter were each repeated seven times, differing only by inputs and outputs. The absence of a capability to specify arrays of graph entities forced the translation into seven unique SPGN files for each function. For large applications with multiple channels of similar processing, this could cause generation of an intractable number of SPGN files. These short comings not withstanding, DIF representations of a representative application designed with Ptolemy was readily translated into a correct SPGN representation which can be realized as executable code with the addition of appropriate Domain Primitives to the Autocoding Toolset's® library.

Section 4 DIF Performance Analysis

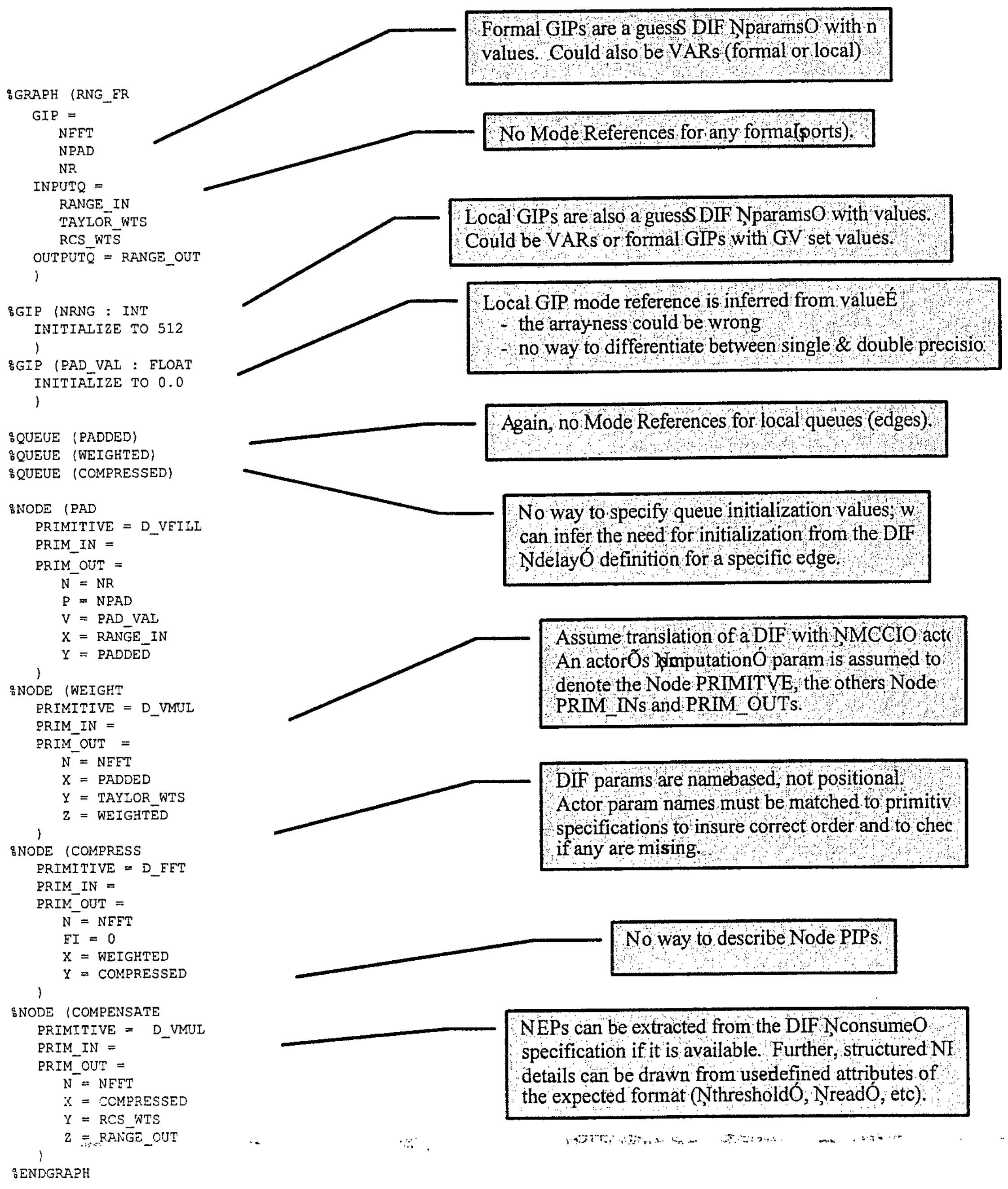
4.1 DIF Feasibility Version Performance Analysis

The application domain for dataflow specifications includes specifying for compilation on single processors and specifying software architectures for parallel architectures. Dataflow specifications contain information not found in control flow specifications (flow chart like specifications) that may be exploited to obtain optimized scheduling and use of memory. To be deterministic, software architectures for processing continuous streams of data must satisfy the Karp and Miller mathematical model for dataflow even if the architectures "virtual circuit" is arrived at through low level, explicit programming of execution synchronization on the architectures processors with transfer of data among them. Dataflow provide a high level means of explicitly specifying architecture level "virtual circuits" of the applications executable components. The University of Maryland's research has been focused on optimizing compilation of dataflow graphs on single processors utilizing dataflow information. MCCI's focus has been on aiding/automating software architecture specifications from systems of dataflow graphs. To be broadly accepted by industry, DIF must be well suited for the full range of application specification.

MCCI undertook the task of analyzing DIF v0.2 capability to efficiently specify HPC applications. While Turing complete, the demonstration version of DIF does not yet contain language features necessary to efficiently specify the full range of applications anticipated. MCCI has implemented a representative set of production applications in the sonar signal processing, radar signal processing, and high performance computing in the course of developing the Autocoding Toolset®. These are applications with large and messy topologies (1000 nodes +), multiple operational modes, and requirements for targeting multiple hardware architectures. Our analysis of DIF focused on engineering convenience features needed by DIF to become and "industrial strength" language capable of efficiently specifying large industrial applications. Additionally, we focused on information needed to translate DIF into strongly typed specifications such as PGM. While information needed for efficient compilation, such as token data types, may be inferred from other elements of the DIF specification, to be broadly acceptable, this information should be made explicit. It can always be ignored if unneeded for a specific target.

PGM is a strongly typed specification requiring either explicit specification of all information needed to determine graph execution behavior (node execution sequences and memory states) and data communication requirements. The DIF feasibility version does not. The first part of our analysis consisted of determining the types of information that had to be added to the PGM specifications translated from DIF. Which the SPGN (textual form of PGM graphs) generated from DIF was completely legal SPGN, not all information needed to generate executable code by the Autocoding Toolset® was included.

This information was added by manual editing. Once added, imported applications autocoded without error. Figure 4.a shows the imported SPGN for the Range subgraph. Yellow comment boxes have been added to the figure to commenting on the additional information needed to autocode the graph into executable code.



```
%GRAPH (RNG_FR
   GIP =
      NFFT
      NPAD
      NR
   INPUTQ =
      RANGE_IN
      TAYLOR_WTS
      RCS_WTS
   OUTPUTQ = RANGE_OUT
   )

%GIP (NRNG : INT
   INITIALIZE TO 512
   )
%GIP (PAD_VAL : FLOAT
   INITIALIZE TO 0.0
   )

%QUEUE (PADDED)
%QUEUE (WEIGHTED)
%QUEUE (COMPRESSED)

%NODE (PAD
   PRIMITIVE = D_VFILL
   PRIM_IN =
   PRIM_OUT =
      N = NR
      P = NPAD
      V = PAD_VAL
      X = RANGE_IN
      Y = PADDED
   )
%NODE (WEIGHT
   PRIMITIVE = D_VMUL
   PRIM_IN =
   PRIM_OUT  =
      N = NFFT
      X = PADDED
      Y = TAYLOR_WTS
      Z = WEIGHTED
   )
%NODE (COMPRESS
   PRIMITIVE = D_FFT
   PRIM_IN =
   PRIM_OUT =
      N = NFFT
      FI = 0
      X = WEIGHTED
      Y = COMPRESSED
   )
%NODE (COMPENSATE
   PRIMITIVE =  D_VMUL
   PRIM_IN =
   PRIM_OUT =
      N = NFFT
      X = COMPRESSED
      Y = RCS_WTS
      Z = RANGE_OUT
   )
%ENDGRAPH
```

Formal GIPs are a guessÑ DIF ÒparamsÓ with n values. Could also be VARs (formal or local)

No Mode References for any formal ports).

Local GIPs are also a guessÑ DIF ÒparamsÓ with values. Could be VARs or formal GIPs with GV set values.

Local GIP mode reference is inferred from valueÉ
- the array-ness could be wrong
- no way to differentiate between single & double precisio

Again, no Mode References for local queues (edges).

No way to specify queue initialization values; w can infer the need for initialization from the DIF ÒdelayÓ definition for a specific edge.

Assume translation of a DIF with ÒMCCIO act
An actorÕs ÒcomputationÓ param is assumed to denote the Node PRIMITVE, the others Node PRIM_INs and PRIM_OUTs.

DIF params are name-based, not positional. Actor param names must be matched to primitiv specifications to insure correct order and to chec if any are mising.

No way to describe Node PIPs.

NEPs can be extracted from the DIF ÒconsumeÓ specification if it is available. Further, structured NI details can be drawn from user-defined attributes of the expected format (ÒthresholdÓ, ÒreadÓ, etc).

Figure 4.a - Imported DIF with Analysis Comments.

instance, it is perfectly legal for there to be no consumption specification. Also, a DIF node with a refinement (indicating subgraph) may also have an actor with a computation and params (indicating node).

Figure 4.b illustrates editing inserting the needed information. SPGN added with manual editing is highlighted in red. Comments on the edits are preceded with %%.

```
%GRAPH (FR_SAR
  INPUTQ =
    SAR_IN : CFLOAT,
    TAYLOR : CFLOAT ARRAY(2048),  %% these 2 items were GIPs in
                          %%source SPGN
    RCS : FLOAT ARRAY(2048),      %% -> changed to Qs due to limitations
                          %%    of DIF "interface" and " params"
                          %%    statements?
    AZKERN : CFLOAT ARRAY(1024)   %% was a VAR in source SPGN
                          %% -> also changed for <above> reason?
  OUTPUTQ = SAR_OUT : CFLOAT
  )
%GIP (NFFT_AZI : INT
  INITIALIZE TO 1024
  )
%GIP (NFFT_RNG : INT
  INITIALIZE TO 2048
  )
%GIP (NFILL : INT
  INITIALIZE TO 16
  )
%GIP (N_R : INT
  INITIALIZE TO 2032
  )
%QUEUE (RNGOUT : CFLOAT
  INITIALIZE TO                    %% initialization (amt &  vals)
  (NFFT_RNG*NFFT_AZI)/2 OF <0.0E0,0.0E0>)  %% lost in DIF
representation
%SUBGRAPH (RANGE
  GRAPH = RNG_FR
•
•
```

```
%GRAPH (RNG_FR
  GIP =
    NFFT : INT,
    NPAD : INT,
    NR : INT
  INPUTQ =
    RANGE_IN : CFLOAT,
    TAYLOR_WTS : CFLOAT ARRAY(2048),
    RCS_WTS : FLOAT ARRAY(2048)
  OUTPUTQ = RANGE_OUT : CFLOAT
  )
%GIP (NRNG : INT
  INITIALIZE TO 512
  )
%GIP (PAD_VAL : CFLOAT          %% PAD_VAL should be
                                %%CFLOAT
  INITIALIZE TO <0.0E0, 0.0E0>
  )
%QUEUE (PADDED : CFLOAT)
%QUEUE (WEIGHTED : CFLOAT)
%QUEUE (COMPRESSED : CFLOAT)
%NODE (PAD
  PRIMITIVE = D_VFILL
  PRIM_IN =
    NR,
    NPAD,
    UNUSED,               %% UNUSED param missing in
DIF
    PAD_VAL,
    RANGE_IN
      THRESHOLD = NRNG*NR      %% no NEPS in DIF
  PRIM_OUT =
•
```

Figure 4.b - Manual Edits to Imported SPGN for SAR and Range Processing Graphs

As can be seen in the example of figure 4.a, data token type and memory state information required to compile an executable with a static memory map was added to the imported SPGN. Called modes in SPGN, token types for queues, variable, and parameters had to be added manually. Initialization was also specified for the queue RNGOUT to insure initial states of queues connecting range and azimuth processes. While DIF is actor independent, UMD used the Ptolemy actor set in their specification. Actors mimicking MCCI domain Primitives were added to the Ptolemy library for the SAR application. However, the actor parameter specifications are name ordered while Domain Primitives are place ordered. An UNUSED had to be added to preserve place ordering in the imported subgraphs. Also Node Execution Parameters (NEPs) not included in the specification of queue inputs to actors had to be added. This is an example of information that may be inferred from Ptolemy actor specifications but must be made explicit in PGM.

4.2 Common Domain Actor Library

4.2 Common Domain Actor Library

The most significant problem encountered in exchanging graphs between programming environments was the lack of a common library of the executable core of dataflow graph nodes. DIF is actor independent, i.e. it specifies dataflow topology not functionality. Functionality comes from the specifications of the actors referenced in the dataflow graph specification. This comment applies to dataflow graphs in general. However to target a specific programming environment, members of the programming environments actor/primitive library must be referenced. This problem was largely avoided in the SAR graph exchange experiment by the addition of actors to the Ptolemy library that mimicked the Domain Primitives of the PGM specification. However examination of the multi rate filter code imported from Ptolemy clearly shows the problem. Inserted editing (high lighted in red) includes actor specifications not included in the Domain Primitive library. Autocoding this graph would require implementation of Domain Primitives mimicking Ptolemy primitives.

The Actor Interchage Format was developed by the UMD team as a potential solution to this actor compatibility problem. Techniques for transferring DSP applications across design tools with a high degree of automation have also been investigated by the UMD team. Our work in this area motivates a new paradigm of porting DSP applications across dataflow-based design tools through the use of interchange specifications. The portability of DSP applications across design tools is equivalent to portability across all underlying embedded processing platforms and DSP code libraries supported by those tools. Such portability would therefore clearly be a powerful capability if it can be attained through a high degree of automation, and a correspondingly low level of manual or otherwise ad-hoc fine-tuning. The key advantage of using a DIF specification as an intermediate state in achieving such efficient porting of DSP applications is the comprehensive representation in the DIF language of functional semantics and component/subsystem properties that are relevant to design and implementation of DSP applications using dataflow graphs.

Except for information pertaining to the detailed functionality and operation of individual actors, a DIF specification for a DSP application should have the same essential "meaning" independent of what particular tool the specification is presented to (assuming the tool supports all of the features required by the application model). Therefore, porting DSP applications through DIF can be achieved by interchanging the tool-dependent actor information along with the tool-independent dataflow information. For this purpose, we have developed an *actor mapping* capability, based on a novel *actor interchange format* (*AIF*), in DIF that provides an infrastructure for automatically converting actor specifications from tool to tool based on the given actor interchange information and conversion protocols.

The actor interchange format is a specification format dedicated to specifying tool-specific information pertaining to actors in dataflow graphs. Its syntax supports:

a. Mapping from a source actor to a target actor, including any optional conditions (e.g., on parameter values) that trigger the particular mapping.

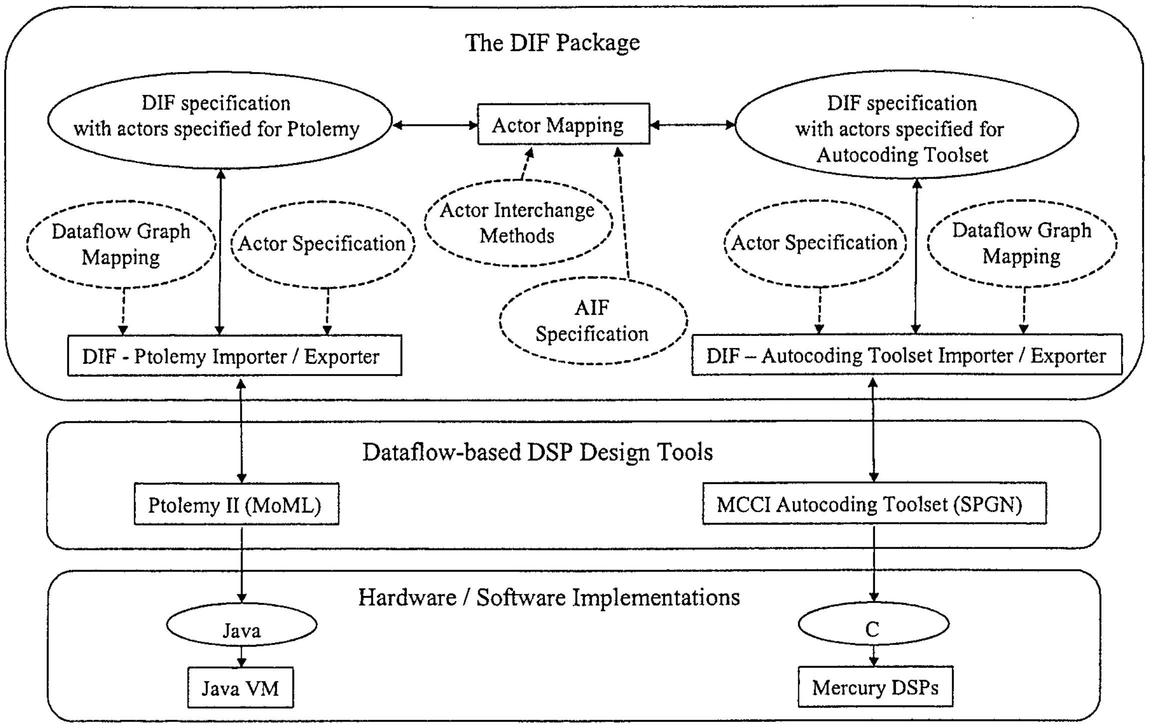
b. Mapping from a source actor to a sub-graph consisting of a set of target actors, including optional triggering conditions.

c. The mapping from a source actor attribute (or set of attributes) to a target actor attribute, including optional specification of an arbitrary-complexity computation for automatically determining the target attribute value from a given set of source attribute values.

Because different design tools usually provide different sets of actor libraries, problems can arise in the porting process due to *actor absence*, *actor mismatch*, and *actor attribute mismatch*. The actor interchange format can significantly ease the burden of actor mismatch problems by allowing a designer a convenient means for making a one-time specification of how multiple modeling components in the target design tool can construct a sub-graph such that the subgraph functionality is compatible to the source actor. In addition to providing automation in the porting process, such conversions reduce the need for users to introduce new actor definitions in the target model, thereby reducing user effort and code bloat.

Similarly, actor interchange methods can solve attribute mismatch problems by evaluating a target attribute in a consistent, centrally-specified manner, based on any subset of source attribute values. For absent actors, most design tools provide ways to create actors through some sort of actor definition language. Once users determine equivalent counterparts for absent and mismatched actors, our actor mapping mechanism can take over the job cleanly and efficiently.

Figure 4.a illustrates the porting mechanism, it consists of three steps: exporting, actor mapping, and importing.



The DIF Package
DIF specification with actors specified for Ptolemy
Actor Mapping
DIF specification with actors specified for Autocoding Toolset
Actor Interchange Methods
Dataflow Graph Mapping
Actor Specification
AIF Specification
Actor Specification
Dataflow Graph Mapping
DIF - Ptolemy Importer / Exporter
DIF – Autocoding Toolset Importer / Exporter
Dataflow-based DSP Design Tools
Ptolemy II (MoML)
MCCI Autocoding Toolset (SPGN)
Hardware / Software Implementations
Java
C
Java VM
Mercury DSPs

Figure 4.a - Porting mechanism through DIF

For a complete introduction of the DIF porting mechanism, including the actor interchange format, and a demonstration example, please refer to Section 7 and Section 8 in "Dataflow Interchange Format Version 0.2" technical report.

While very helpful in mapping between primitive/actor libraries in different environments, the Actor Interface Format does not address the combinatorial explosion that will occur if we are successful in having DIF widely accepted by many tool vendors. Potentially, every tools library would have to be mapped into every other library. The need for including a functional library specification in the full performance version of DIF became clear during the feasibility investigation. At a minimum, this will reduce the potential all to all mapping problem to an all to one problem. However, if functional specifications of an existing and broadly accepted standard primitive library are used as the DIF common Domain Actor library, the problem may be avoided altogether. The VSIPL is the obvious candidate for this standard. VSIPL provides a standard math library supporting code exchange between different vendors processors. The VSIPL standard includes a functional specification for each math library primitive, a calling specification, and a reference implementation. Vendors supporting VSIPL may implement their own performance version of the library routines utilizing the specification and calls or compile the reference implementation. A dataflow graphical equivalent of this approach will support a VSIPL based common Domain Actor library for DIF. Domain actors with VSIPL functional specifications and VSIPL calling specifications incorporated in the Domain Actor specifications may be implemented in target environments by mapping the Domain Actor to an equivalent in the target environment or importing the reference implementation. Meeting the VSIPL requirement to produce computational results identical to the reference primitives can be met while preserving the capability to optimize performance using the unique information in dataflow graph specifications.

4.3 DIF Feasibility Capabilities Analysis

MCCI also analyzed the feasibility version of DIF from the prospective of efficiently specifying typical very large applications targeting parallel architectures. While Turing complete, the feasibility version of DIF lacks certain capabilities necessary to developing compact, reusable specifications for large production codes. Engineering convenience features are needed to avoid specification for "in-line" graph topology for multi-channel systems and repetitive SPGN files for like subgraphs. Features supporting specification of multi mode, reusable specifications are needed in the full performance version of DIF. These features include:

a. Specification of Token Data Types; edges, parameters, interface.

The full performance version of DIF should include the capability to specify data token types for all supported forms of memory specifications. This includes the graphs edges conveying data between graph nodes. These will either specify executable code memory states or inter-processor data transfers. In either case, memory size requirements must be specified.

b. Arrays of entities; nodes, edges, parameters.

While unnecessary for graphs targeting single processors and single channel applications, specification of arrays of entities is essential to compact specification of large multi-channel applications. For example modern submarine sonar beamforming applications have on the order of 1000 channels of essentially identical processing, differing only by inputs, outputs and parameters governing beam steering. Specifying each beams channel independently as currently required would produce intractably complex DIF and SPGN file systems. Autocoding each independent file would severely complicate the task of autocoding the application with ours of other tools. It is considered highly advantages to add an entity array specification capability to DIF,.

c. Complete interface; parameters as well as edges

Ptolemy graphs are intended to execute singly on data included in existing data files. Production applications must include multiple graphs which may be made active, or inactive, in combinations. Multiple copies of the same graph, with each instance parameterized or sourced differently must be supported. Data sources and sinks must include a number of devices and software structures including other graphs. Controls must exist to modify application configuration, data sources and sinks, internal dataflow, and graph functional behavior "on the fly". All aspects of application configuration and behavior must be capable of change by external control while deterministic execution is maintained. For this reason, DIF must be expanded to include support for complete external interfaces, i.e. edges, parameters, and run time variable controls.

d. Edge initialization

It is often convenient to specify initial graph state as tokens on internal queues. These tokens may or may not have to have values assigned. If graphs are not initialized, the behavior model for partitions created by the user that include uninitialized queues may have a transient component to their behavior model. Unless necessary for some functional reason, this unnecessarily complicates code generation for its realizations. If the queues are contained in a circuit, lack of initialization may block the graph creating an illegal and untranslatable condition. Initializing with tokens with values left unassigned may create a unwanted transient in the values of output data even though the behavior model is valid. The capability to specify edge initialization in the full performance version of DIF is recommended.

e. More sophisticated edge behavior

(1) threshold, read, offset read data

While only produce and consume amounts must be specified for edge source and sink to fully a scheduling behavior model for dataflow graphs, it is desirable to ad the capability to specify more sophisticated edge behavior in the full performance version of DIF. Independent specification of Threshold and Read edge parameters allows the intentional introduction of lags between node executions in parallel data streams or to accumulate data to allow primitive functions to iterate execution over a large data set. Specification of a data offset amount further increases node execution flexibility. For example, multiple parameter sets may be passed to nodes via queues and the offset parameter used to select the correct set.

(2) offset, read, consume expression support for variable characteristics

Support for expressions for edge parameters, except threshold, support a wide range of external controls of graph execution behavior and source and sink node functional behavior. Dataflow may be modified on the fly by edge controls either computed by the graph or externally set. Dynamic application behavior is readily specifiable with this capability. Making threshold variable has proved problematic. The issue is synchronizing its computation with node execution, an inherent race condition. MCCI has never come up with a practical way to support run time variable thresholds even when doing so would simplify specifications. This is different than specifying enumerated sets of threshold values at translation time and switching logic to switch between code segments realizing behavior models for each enumerated value.

(3) Dangling edges; no sink and/or source

The capability to leave either the head or tail of an edge dangling, i.e. unconnected has proved highly useful. Leaving an edge tail unconnected but connecting the head in effect creates a private variable for the involved node. While a variable could be used for this purpose, reading data from queues in much faster than reading from variable. These queues are typically assigned a 0 value for threshold and consume, a read amount appropriate for the node function, and possible variable offset values to select among multiple data sets assigned to the edge during initialization.

The capability to leave edge heads unconnected provides a convenient means to dump data in the "bit bucket" if temporarily unneeded. Used in conjunction with external controls governing inter graph connectivity, it provides a convenient means to drop data from inactive elements of the application.

f. More sophisticated node execution behavior

Node execution models should be extended to include a control interface between inputs and the underlying primitive. This control interface should be capable of sending and receiving synchronization signals (the equivalent of semaphores) and setting and evaluating expressions associated with edge parameters and primitive controls. The interfaces should have a capability to perform simple arithmetic calculations as well. This capability supports enhancing the flexibility of dataflow graph execution. For example, the synchronization of reading and writing graph variables with the execution of nodes using variable values has long been problematic for pure dataflow environments. The addition of control interfaces supporting synchronization signals to nodes allows enforcing executions that will ensure deterministic assignment of values to variables before the values are used.

g. Multi-mode graphs

Production codes for modern weapons systems are frequently multi modal. Applications must be seamlessly reconfigurable among execution many execution behaviors. The ability to specify multiple graph behavior modes is considered essential to the full performance version of DIF

(1) Production control; variable output and "valving"

The capability to vary production amounts by either expressions evaluated at node execution of on/off controls set by external control or internally computed variables provides the capability to partition dynamic dataflow applications into systems of static dataflow partitions. Nodes with variable production amounts of valves must be on the edge of partitions. This capability allows the use of the full range of static behavior models for efficient translation to executable code while maintaining dynamic behavior at the application level to support data dependent functionality and application reconfiguration.

(2) Parameter characteristics; Static vs. variable values, value "sets"

Application may frequently be faced with the conflicting need to make parameters governing graph execution behavior static to enable translation to control flow executable realizations while at the same time support multiple sets of parameters for different functional behavior. Unfortunately the different parameter sets impose different execution behaviors. In order to have this both ways, parameter sets may be enumerated and multiple behavior models created matching parameter set values. Realizations may include multiple code sections, one per behavior, and switching logic to select the appropriate segment depending on the value set in use at node execution.

If variables affect only functional behavior, i.e. data token values only, they may be made variable in a set of otherwise fixed value parameter sets.

(3) Hidden graph elements

Hidden graph elements is a concept currently under investigation that may be a generalized approach to modifying active graph topology under external control that is the equivalent of changing valve control value sets. Its use to accomplish one or more of the above requirements for the full performance version of DIF should be pursued.

Section 5 Study Conclusions

5.1 Study Results

Phase I of this STTR was a study to determine the feasibility of developing a common, vendor neutral dataflow language that can be supported by all dataflow tools and programming environments as a standard for dataflow graph exchange. Our working hypothesis is that Dataflow Interchange Format, DIF, initiative undertaken at the University of Maryland could be developed as such a standard. Our investigation was partitioned into two areas:

a. Development of a feasibility version of DIF, and

b. Demonstration of the feasibility of graph exchange between different dataflow programming environments using DIF.

DIF version 0.2 was developed by the University of Maryland as an expansion of the exploratory version 0.1. DIF has been shown to be a Truing complete dataflow language. Prototype tool support was developed for DIF in MCCI's Autocoding Toolset® to export and import DIF specifications from and to PGM graphs in the notational SPGN format. Corresponding tools were developed or extended by the University investigators to import and export DIF graphs into and from Ptolemy data structures. MCCI's Autocoding Toolset® and the University of California's Ptolemy, now used by the University of Maryland in their work, were independently developed from a common mathematical model. MCCI's tools use a language representation developed by the Navy which has spawned a number of commercial dataflow tools and environments. Likewise the Ptolemy work has inspired a number of dataflow research efforts in other universities and laboratories. Development of a notational form of dataflow graph specification that can be supported by both environments is very clear indication of its general supportability by all dataflow tools and environments. It is likely that some language features planned for a full performance version of DIF may not be necessary or supported for some environments. Unneeded information in DIF specifications can be ignored without invalidating the remaining specification. And proper subsets of DIF specifications can be imported and expanded in the destination environment.

It is considered that development of DIF v0.2 with associated prototype tool support in independent dataflow programming environments demonstrates the feasibility of a common, vendor neutral dataflow language is feasible. A full performance version expanding the prototype with features for specifying more complex applications can be well suited for adoption as an industry standard.

The second part of our effort was demonstrating porting applications between Ptolemy and Autocoding Toolset® environments using DIF as a common specification language. These experiments were described in section 3.

A synthetic aperture radar benchmark, common to high performance embedded computing, was implemented in PGM using the Autocoding Toolset®. PGM specifications were translated to DIF. The DIF specifications were translated to Ptolemy data structures and the application executed on a Ptolemy platform. DIF specifications were generated from the Ptolemy implementation. The DIF specification of the Ptolemy version was imported back into the Autocoding Toolset® and translated to SPGN. The imported version was autocoded to an executable realization. Comparison of the execution of the original benchmark, the imported Ptolemy version, and the reimported Autocoding Toolset® version identical results differing only by precision level errors. This round trip experiment demonstrates the graph exchange that DIF is designed to support.

A second demonstration was conducted importing multi-rate filter Ptolemy benchmark into the Autocoding Toolset® environment. DIF specifications were generated from the Ptolemy benchmark. SPGN specifications for the Autocoding Toolset® were translated from the DIF using the prototype DIF import tool. Correct SPGN was generated. This experiment was not continued through realization on an executable version because the effort required to develop primitives in the Autocoding Toolset® corresponding to Ptolemy primitives would have exceeded phase I scope. However, the ability to generate correct SPGN is clear confirmation of the capability to import Ptolemy designs.

The imported SPGN nodes referenced Ptolemy primitives, highlighting a challenge for realizing DIF as an industry standard. Domain Primitive support can be developed for Ptolemy and visa versa. An actor interchange format tool was developed to assist in primitive/actor mapping. However, a more general approach to achieving common functional specifications within the common dataflow behavior specified by DIF is considered essential to developing a widely acceptable dataflow language standard. The VSIPL primitive library standard can readily be adopted as the standard functional specification for a DIF Domain Actor library. Nodes in the DIF specification can specify the VSIPL standards functionality and with the standards calling structure. Functional performance may be specified by the VSIPL reference library while individual vendors can implement performance versions specific to their environments. This approach to using VSIPL as a library standard for porting between dataflow environments will add a new capability to VSIPL portability.

During the course of the feasibility study, a number of language features which are desirable for compact specification for large and complex applications were identified. DIF v0.2 was found to be Turing complete. These "engineering convenience" features identified are unnecessary for determinism, but will significantly enhance the user friendliness of the language. Suggested new capabilities were discusses in section 4. The MCCI/UMD team believes that DIF can readily be expanded to include these features and indeed work to include them is already in progress is many cases.

Appendix A to Final Report on

DIF - A Language for Dataflow Graph Specification and Exchange

October 27, 2004

Sponsored by

Defense Advanced Research Projects Agency (DOD)
(Controlling DARPA Office)

ARPA Order C043/70

Issued by U.S. Army Aviation and Missile Command Under

DAAH01-03-C-R236

Preliminary version of the Dataflow Interchange Format Version 0.2 language specification document (Technical Report, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2004, to appear).

Dataflow Interchange Format Version 0.2

Chia-Jui Hsu and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{jerryhsu, ssb}@eng.umd.edu

Abstract

The Dataflow Interchange Format (DIF) is a standard language to specify mixed-grain dataflow models for digital signal, image, and video processing (DSP) systems and other streaming-related application domains. Major objectives of the DIF project are to design this standard language; to provide an extensible repository for representing, experimenting, and developing dataflow models and techniques; and to facilitate technology transfer of applications across DSP design tools. The first version of DIF [8, 12] has demonstrated significant progress towards these goals. The subsequent phase of the DIF project, which we discuss in this report, is focusing on improving the DIF language and the DIF package to represent more sophisticated dataflow semantics and exploring the capability of DIF in transferring DSP applications and technology. This exploration has resulted so far in an approach to automate exporting and importing processes and a novel solution to porting DSP applications through DIF. This report introduces the DIF language version 0.2 along with the DIF package, the supported dataflow models, the approach to exporting and importing, and the newly proposed porting mechanism.

1 Introduction

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models [1, 4, 5, 6, 13] have been developed for dataflow-based design. Nowadays, a growing set of DSP design tools support such dataflow semantics [3]. A critical issue arises in transferring technology across these design tools due to the lack of a standard and vendor-independent language and an associated package with intermediate representations and efficient implementations of dataflow analysis and optimization algorithms. DIF is designed for this purpose and is proposed to be a standard language for specifying and working with dataflow-based DSP applications across all relevant dataflow modeling approaches that are relevant to DSP system design.

The first version of DIF demonstrated partial success in achieving such goals. However, its language syntax and accompanying intermediate representations are insufficient to handle more complicated dataflow models as well as to transfer DSP applications through significant levels of automation. Therefore, in the subsequent phase of our work on DIF, version 0.2 of the DIF language was developed and the associated DIF package was also extended to address these challenges. With the enhanced DIF language and DIF package, advanced dataflow modeling techniques such as parameterized synchronous dataflow [1] and boolean-controlled dataflow [6] can be fully supported.

In order to provide the DSP design industry with a convenient front-end to use DIF and the DIF package, automating the exporting and importing processes between DIF and design tools is an essential feature. Although the problems in exporting and importing are design-tool-specific, practical implementation issues are quite common among different design tools. These issues have been carefully studied and we describe our approaches to addressing them in this report.

The problem of transferring DSP applications across design tools with a high degree of automation has also been investigated. Such porting typically requires tedious effort, is highly error-prone, and is very fragile with respect to changes in the application model being ported (changes to the model require further manual effort to propagate to the ported version). This motivates a new approach to porting DSP applications across dataflow-based design tools through the interchange information captured by the DIF language, and through additional infrastructure and utilities to aid in conversion of complete dataflow-based application models (including all dataflow- and actor-specific details) to and from DIF.

Portability of DSP applications across design tools is equivalent to portability across all underlying embedded processing platforms and DSP code libraries supported by those tools. Such portability would clearly be a powerful capability if it can be attained through a high degree of automation, and a correspondingly low level of manual or otherwise ad-hoc fine-tuning. The key advantage of using a DIF specification as an intermediate state in achieving such efficient porting of DSP applications is the comprehensive representation in the DIF language of functional semantics and component/subsystem properties that are relevant to design and implementation of DSP applications using dataflow graphs.

The organization of this report is as follows. In Section 2, we review the basic concepts of dataflow graphs and introduce the structure of DIF. Then we describe DIF language version 0.2 in Section 3. Next, we illustrate dataflow models in DIF through examples of DIF specifications in Section 4. In Section 5, we introduce the DIF package and discuss the overall methodology of design using DIF. In Section 6, we discuss critical problems in exporting and importing DIF specifications from DSP design tools and describe our approaches to these problems. In Section 7, we develop the porting mechanism of DIF and discuss associated actor mapping issues. In Section 8, we show the feasibility of the DIF porting capabilities by demonstrating the porting of a Synthetic Aperture Radar application from the MCCI Autocoding Toolset [15, 16] to Ptolemy II [9, 10, 11]. In the final section, we list some major directions for future work.

2 Dataflow Graphs and Hierarchical Dataflow Representation

2.1 Dataflow Graph

In the dataflow modeling paradigm, computational behavior is depicted as a dataflow graph (DFG). A dataflow graph G is an ordered pair (V,E), where V is a set of vertices, and E is a set of directed edges. A directed edge $e = (v1,v2) \in E$ is an ordered pair of a source vertex $src(e)$ and a sink vertex $snk(e)$, where $src(e) \in V$, $snk(e) \in V$, and e can be denoted as $v1 \rightarrow v2$. Given a directed graph $G = (V, E)$ and a vertex $v \in V$, the set of incoming edges of v is denoted as $in(v) = \{e \in E | snk(e) = v\}$, and similarly the set of outgoing edges of v is denoted as $out(v) = \{e \in E | src(e) = v\}$.

In dataflow graphs, a vertex v (also called a *node*) represents a computation and is often associated with a node *weight*. The weight of an object in DIF terminology refers to arbitrary information that a user wishes to associate with the object (e.g., the execution time of a node or the type of data transferred along an edge). An edge e in dataflow graphs is a logical data path from its source node to its sink node. It represents a FIFO (first-in-first-out) queue that buffers data values (tokens) for its sink node. An edge has a non-negative integer delay $delay(e)$ associated with it and each delay unit is functionally equivalent to a z^{-1} operator.

Dataflow graphs naturally capture the data-driven property in DSP computations. An actor (node) can fire (execute) at any time when it is *enabled* (the actor has sufficient tokens on all its incoming edges to perform a meaningful computation). When firing, it consumes certain numbers of tokens from its incoming edges $in(v)$, executes the computation, and produces certain numbers of tokens on its outgoing edges $out(v)$. This combination of consumption, execution, and production may or may not be carried out in an interleaved manner. Given an edge $e = (v1,v2)$, in a dataflow graph, if the the number of tokens produced on e by an invocation of $v1$ is constant throughout execution of the graph, then this constant number of tokens produced is called the *production rate* of e and is denoted by $prd(e)$. The *consumption rate* of e is defined in an analogous fashion, and this rate, when it exists, is denoted by $cns(e)$.

2.2 Hierarchical Structure

In dataflow-based DSP systems, the granularity of actors can range from elementary operations such as addition, multiplication, or logic operations to DSP subsystems such as filters or FFT algorithm. If an actor represents an indivisible operation, it is called *atomic*. An actor that represents a hierarchically-nested subgraph is called a *supernode;* an actor that does not is called a *primitive node*. The *granularity* of a dataflow actor describes its functional complexity. Simple primitive actors such as actors for addition or for elementary logic operations are called *fine-grained* actors. If the actor complexity is at the level of signal processing sub-tasks, the actor is called *coarse-grained.* Practical dataflow models of applications typically contain both fine- and coarse-grained actors; dataflow graphs underlying such models are called *mixed-grain* graphs.

In many sophisticated DSP applications, the mixed-grain dataflow graph of an overall system consists of several supernodes, and each top-level supernode can be further refined into another mixed-grain dataflow graph, possibly with additional (nested) supernodes.

One way to describe such complicated systems is to flatten the associated hierarchies into a single non-hierarchical graph that contains no supernodes. However, such an approach may not always be useful for the following reasons. First, analyzing a dataflow graph with the original hierarchical information intact may be more efficient than trying to analyze an equivalent flattened graph that is possibly much larger. Second, the top-down design methodology is highly applicable to DSP system design, so the overall application is usually most naturally represented as a hierarchical structure. Thus, incorporating hierarchy information into the DIF language and graph representations is an essential consideration in the DIF project.

Definitions related to hierarchies are introduced as follows. A *supernode* s in a graph $G = (V, E)$ represents a dataflow subgraph G', and this association is denoted as $s \cong G'$. The collection of all supernodes in G forms a subset S in V such that $s \in S \subset V$ and $\forall v \in \{V - S\}$, v is a primitive node. If a supernode s in G represents the nested graph G', then G' is a *subgraph* of G and G is the *supergraph* of G'.

A *hierarchy* $H = (G, I, M)$ contains a graph G with an interface I, and a mapping M. Given another hierarchy $H' = (G', I', M')$, if G' is a subgraph of G, we said that H' is a sub-hierarchy of H and H is a super-hierarchy of H'.

A *mapping* from a supernode s representing subgraph G' to a sub-hierarchy H' containing G' is denoted as $s \Rightarrow H'$, where $s \cong G'$ and $H' = (G', I', M')$. The *mapping* M in a hierarchy $H = (G, I, M)$ is a set containing all mappings (to subhierarchies) of supernodes s in $G = (V, E)$; that is, $\forall s \in S \subset V$, $\{s \Rightarrow H'\} \in M$.

The *interface* I in hierarchy H is a set consisting of all interface ports in H. An *interface port* (or simply called *port*) p is a dataflow gateway through which data values (tokens) flow into a graph or flow out of a graph. From the interior point of view, a port p can associate with one and only one node v in graph G, and this association is denoted as $p{:}v$, where $p \in I$, $v \in V$, $G = (V, E)$ and $H = (G, I, M)$. From the exterior point of view, a port p can either connect to one and only one edge e'' in graph G'' or connect to one and only one port p'' in hierarchy H'', where G'' is the supergraph of G and H'' is a super-hierarchy of H. These connections are denoted as $p \sim e''$ and $p \sim p''$ respectively, where $p \in I$, $e'' \in E''$, $p'' \in I''$, $H = (G, I, M)$, $G'' = (V'', E'')$, and $H'' = (G'', I'', M'')$.

An interface port is directional; it can either be an *input port* or an *output port*. An input port is an entry point for tokens flowing from outside the hierarchy to an inside node, and conversely, an output port is an exit point for tokens moving from an inside node to somewhere outside the hierarchy. Given $H = (G, I, M)$, $in(I)$ denotes the set of input ports of H and $out(I)$ denotes the set of output ports of H, where $in(I) \cap out(I) = \varnothing$, and $in(I) \cup out(I) = I$. Then given a port $p \in in(I)$, $p{:}v$, and $p \sim e''$, v consumes tokens from e'' when firing. Similarly, given $p \in out(I)$, $p{:}v$, and $p \sim e''$, v produces tokens to e'' when firing.

The association of an interface port with an inside node and the connection of an outer edge to an interface port can facilitate the clustering and flattening processes. For example, given $p{:}v$, $p \sim e''$, $src(e'') = v''$, $snk(e'') = s$, $s \Rightarrow H = (G, I, M)$, and $p \in I$, a new edge e can be connected from v'' to v directly after flattening the hierarchy H.

With the formal dataflow graph definition reviewed in Section 2.1 and the hierarchical structures defined in this section, we are able to precisely specify hierarchical dataflow graphs in the DIF language, which is introduced in the following section.

3 The DIF Language

The Dataflow Interchange Format (DIF) is proposed to be a standard language for specifying dataflow semantics in dataflow-based application models for DSP system design. This language is suitable as an interchange format for different dataflow-based DSP design tools because it provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification.

From the dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information. The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool. Therefore, the dataflow semantics of a DSP application is unique in DIF regardless of any design tool used to originally enter the application specification. Moreover, DIF also provides syntax to specify design-tool-specific information, and such tool-specific information is captured within the data structures associated with the DIF intermediate representations. Although this information may be irrelevant to many dataflow-based analyses, it is essential in exporting, importing, and transferring across tools, as well as in code generation.

DIF is not aimed to directly describe detailed executable code. Such code should be placed in actual implementations, or in libraries that can optionally be associated with DIF specifications. Unlike other description languages or interchange formats, the DIF language is also designed to be read and written by designers who wish to specify DSP applications in dataflow graphs or understand applications based on dataflow models of computations. As a result, the language is clear, intuitive, and easy to learn and use for those who have familiarity with dataflow semantics.

DSP applications specified by the DIF language are referred to as *DIF specifications*. The DIF package includes a frond-end tool, the DIF language parser, which converts a DIF specification into a corresponding graph-theoretic intermediate representation. This parser is implemented using a Java-based compiler-compiler called SableCC [7]. The complete SableCC grammar of the Dataflow Interchange Format is presented in Appendix A.

3.1 Dataflow Interchange Format Language Version 0.2

The first version of the DIF language [8], version 0.1, was the first attempt to approach aforementioned goals. In DIF version 0.1, we demonstrated the capability of conveniently specifying and manipulating fundamental dataflow models such as SDF and CSDF. Nonetheless, its semantics is insufficient to describe in detail more advanced dataflow semantics and to specify actor-specific information. As a result, the DIF language has been further developed to the second version, version 0.2, for supporting an additional set of important dataflow models of computation and facilitating design-tool-dependent transferring processes.

Note that any dataflow semantics can be specified using the "DIF" model of dataflow supported by DIF, and the corresponding DIFGraph intermediate representation, however, for performing sophisticated analyses and optimizations for a particular dataflow model of computation it is usually useful to have more detailed and customized features in DIF that support the model. This is why the exploration of different dataflow models for incorporation into DIF is an ongoing area for further development of the language and software infrastructure.

From version 0.1 to version 0.2, the syntax consistency and code reusability support of DIF have been improved significantly. DIF language version 0.2 also supports more flexible parameter assignment and provide more flexible treatment of graph attributes. Moreover, it supports most commonly used value types in DSP applications and provides arbitrary naming spaces. Also, perhaps most significantly, the *actor* block is newly created in DIF version 0.2 for specifying design-tool-dependent actor information.

DIF version 0.2 consists of eight blocks: basedon, topology, interface, parameter, refinement, built-in attribute, user-defined attribute, and actor. Those blocks specify different aspects of dataflow semantics and modeling information. The following subsections introduce the syntax of the DIF language.

3.2 The Main Block

A dataflow graph is specified in the main block consisting of two arguments, *dataflowModel* and *graphID*[1], followed by the main braces. The *dataflowModel* keyword specifies the dataflow

1.

model of the graph. The *graphID* specifies the name (identifier) of the graph. The following is the overall of the main block:

```
dataflowModel graphID {
   basedon { ... }
   topology { ... }
   interface { ... }
   parameter { ... }
   refinement { ... }
   builtInAttr { ... }
   attribute usrDefAttrID { ... }
   actor nodeID { ... }
}
```

The eight blocks are defined in the main braces. Each block starts with a block keyword and the content is enclosed by braces. Statements inside block braces end with semicolons. Conventionally, identifiers in DIF only consist of alphabetic, underscore, and digit characters. However, DIF also supports arbitrarily-composed identifiers by enclosing them between two dollar-sign characters. The basedon, topology, interface, parameter and refinement blocks should be defined in this particular order. Except the topology block, however, all other blocks are optional. A top-level graph specification does not have to define the interface block.

3.3 The Basedon Block

```
basedon { graphID; }
```

The *basedon* block provides a convenient way to refer to a pre-defined graph, which is specified by *graphID*. As long as the referenced graph compatible topology, interface, and refinement blocks, designers can simply refer to it and override the name, parameters and attributes to instantiate a new graph. In many DSP applications, duplicated subgraphs usually have the same topologies but different parameters or attributes. The basedon block is designed to support this characteristic and promote conciseness and code reuse.

3.4 The Topology Block

```
topology {
   nodes = nodeID, ..., nodeID;
   edges = edgeID (sourceNodeID, sinkNodeID),
           ...,
           edgeID (sourceNodeID, sinkNodeID);
}
```

The *topology* block specifies the topology of a dataflow graph $G = (V, E)$. It consists of a node definition statement defining every node $v \in V$ and an edge definition statement defining every edge $e = (v_i, v_j) \in E$.

The keyword *nodes* is the keyword for a node definition statement and node identifiers, *nodeIDs*, are listed following the keyword and equals sign. Similarly, *edges* is the keyword for an edge definition statement and edge definitions are listed in a similar fashion. An edge definition, *edgeID (sourceNodeID, sinkNodeID)*, consists of three arguments in order to specify a directed edge: the edge identifier *edgeID*, the source node identifier *sourceNodeID*, and the sink node identifier *sinkNodeID*.

3.5 The Interface Block

```
interface {
   inputs = portID : assocNodeID, ..., portID : assocNodeID;
   outputs = portID : assocNodeID, ..., portID : assocNodeID;
}
```

The *interface* block defines the interface I of a hierarchy $H = (G, I, M)$. An input definition statement defines every input port $p_i \in in(I)$ and the corresponding inside association $p_i : v_i$. Similarly, an output definition statement defines every output port $p_o \in in(I)$ and the corresponding inside association $p_o : v_o$, where $v_i, v_o \in V$ and $G = (V, E)$.

The keywords *inputs* and *outputs* are the keywords for input and output definition statements. Following the *inputs* or *outputs* keyword, port definitions are listed. A port definition, *portID : assocNodeID*, consists of two arguments, a port identifier and its associated node identifier. DIF permits defining an interface port without an associated node, so *assocNodeID* is optional.

3.6 The Parameter Block

```
parameter {
   paramID = value;
   paramID : range;
   paramID;
   ...,
}
```

In many DSP applications, designers often parameterize important attributes such as the frequency of a sine wave generator and the order of a FFT actor. In interval-rate, locally-static dataflow [17], unknown production and consumption rates are specified by their minimum and maximum values. In parameterized dataflow [1], production rate and consumption rates are even allowed to be unspecified and dynamically parameterized. The *parameter* block is designed to support parameterizing values in ways like these, and to support value ranges, and value-unspecified attributes.

In a parameter definition statement, a parameter identifier *paramID* is defined and its *value* is optionally specified. DIF supports various value types and those types are introduced in Section 3.11.

DIF also supports specifying the range of possible values for a parameter. The *range* is specified as an interval such as (1, 2), (3.4, 5.6], [7, 8.9), [-3.1E+3, +0.2e-2], or a set of discrete numbers such as {-2, 0.1, +3.6E-9, -6.9e+3}, or a combination of intervals and discrete sets such as (1, 2) + (3.4, 5.6] + [7, 8.9) + {-2, 0.1, +3.6E-9, -6.9e+3}.

3.7 The refinement block

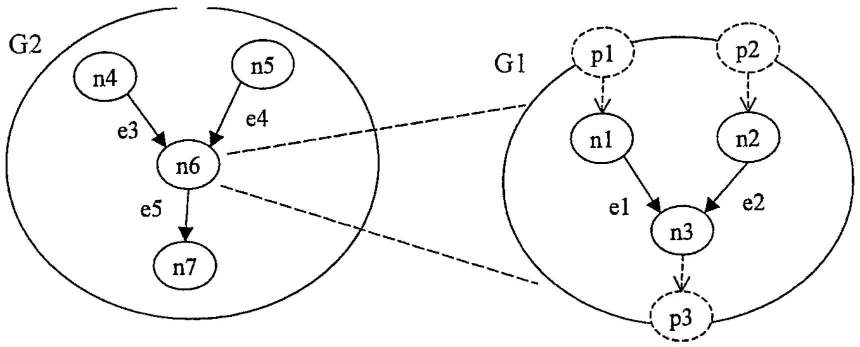
```
refinement {
   subgraphID = supernodeID;
   subportID : edgeID;
```

```
    subportID : portID;
    subParamID = paramID;
    ...;
}
```

The *refinement* block is used to represent hierarchical graph structures. For each supernode $s \in S \subset V$ in a graph $G = (V, E)$, there should be a corresponding refinement block in the DIF specification to specify the supernode-subgraph association, $s \Rightarrow H'$. In addition, for every port $p' \in I'$ in sub-hierarchy $H' = (G', I', M')$, the connection $p' \sim e$, or $p' \sim p$ is also specified in this refinement block, where $e \in E$, $p \in I$, $H = (G, I, M)$, and H is the super-hierarchy of H'. Moreover, the unspecified parameters (parameters whose values are unspecified, e.g., because they may be unknown in advance or computed at runtime) in subgraph G' can also be specified by parameters in G.

Each refinement block consists of three types of definitions. First, a subgraph-supernode refinement definition, *subgraphID = supernodeID*, defines $s \cong G'$. Second, subgraph interface connection definitions, *subportID : edgeID* or *subportID : portID*, describe $p' \sim e$ or $p' \sim p$. Third, a subgraph parameter specification, *subParamID = paramID*, specifies blank parameters in the subgraph by using parameters defined in the current graph.

Figure 1 illustrates how to use DIF to specify hierarchical dataflow graphs. In Figure 1, there are two dataflow graphs, *G1* and *G2*, and supernode *n6* in graph *G2* represents the subgraph *G1*. The corresponding DIF specification is also presented in Figure 1.



```
dif graph G1 {
  topology {
    nodes = n1, n2, n3;
    edges = e1 (n1, n3), e2 (n2, n3);
  }
  interface {
    inputs = p1 : n1, p2 : n2;
    outputs = p3 : n3;
  }
}
```

```
dif graph G2 {
  topology {
    nodes = n4, n5, n6, n7;
    edges = e3 (n4, n6),
            e4 (n5, n6), e5 (n6, n7);
  }
  refinement {
    G1 = n6;
    p1 : e3; p2 : e4; p3 : e5;
  }
}
```

Figure 1. Hierarchical graphs and the corresponding DIF specifications.

3.8 The Built-in Attribute Block

```
builtInAttrID {
   elementID = value;
   elementID = ID;
   elementID = ID1, ID2, ..., IDn;
}
```

The keyword *builtInAttrID* points out which built-in attribute is specified. The element identifier, *elementID*, can be a node identifier, an edge identifier, or a port identifier to which the built-in attribute belongs. It can also be left blank; in this case, the built-in attribute belongs to the graph itself. DIF supports assigning attributes by a variety of value types, an identifier, or a list of identifiers. The supported value types are introduced in Section 3.11.

Usually, the built-in attribute block is used to specify dataflow modeling information. Every dataflow model in DIF can define its own built-in attributes and its own method to process those built-in attributes. The DIF language parser treats built-in attributes in a special way such that the method defined in the corresponding parser is invoked to handle them. Some dataflow models require model-specific attributes and value types, and DIF specifications for those dataflow models will be discussed in Section 4.

In general, *production*, *consumption*, and *delay* are commonly-used built-in attributes of edges in many dataflow models. For example, if *delay(e1) = 1D* and *delay(e2) = 2D*, where *D* is a delay unit, the delay attribute block is specified as: `delay { e1 1; e2 2; }`. Note that the built-in attributes *production* and *consumption* are not exclusive to edges. In hierarchical dataflow models, the interface-associated node has no edge on the corresponding direction. In such cases, specifying production rates or consumption rates as port attributes is permitted in DIF.

3.9 The User-Defined Attributes Block

```
attribute usrDefAttrID {
   elementID = value;
   elementID = ID;
   elementID = ID1, ID2, ..., IDn;
}
```

The user-defined attributes block allows designers to define and specify their own attributes. The syntax is the same as the built-in attributes block. The only difference is that this block starts with the keyword *attribute* followed by the user-defined attribute identifier, *usrDefAttrID*.

3.10 The Actor Block

```
actor nodeID {
   computation = "stringDescription";
   attributeID : attributeType = value;
   attributeID : attributeType = ID;
   attributeID : attributeType = ID1, ID2, ..., IDn;
}
```

The topology, interface, parameter, refinement and built-in attribute blocks are used to describe graph topology, hierarchical structure, and dataflow semantics. They are sufficient for applying dataflow-based analysis and optimization techniques. However, in order to preserve the functionality of DSP applications in design tools, the information supported in DIF language version 0.1 is not enough. As a result, the actor block has been created in DIF language version 0.2 to specify tool-specific actor information.

The keyword *actor* is used for the actor block. The associated *computation* is a built-in actor attribute for specifying in some way the actor's computation (what the actor does). Other actor information is specified as attributes. Explicitly, the identifiers of actor's components such as ports, arguments, or parameters are used as *attributeID* in the DIF actor block. Moreover, the type of the component can be optionally specified as *attributeType*. DIF supports three built-in actor attribute types: INPUT, OUTPUT, and PARAMETER to indicate the interface connections and parameters of an actor. Attributes can be assigned a value, or an identifier for specifying its associated element (edge, port, or parameter), or a list of identifiers for indicating multiple associated elements of the attribute.

The actor block is primarily used in exporting and importing DIF as well as porting DSP applications. Section 6 and Section 7 contain more explanations and examples of how to use the DIF actor block.

3.11 The Value Types

DIF version 0.2 supports most commonly used value types in DSP operations: integer, double, complex, integer matrix, double matrix, complex matrix, string, boolean, and array. Scientific notation is supported in DIF in the double format. For example, a double value can have the following formats: 123.456, +0.1, -3.6, +1.2E-3, -4.56e+7. A complex value is enclosed by parentheses as (real part, imaginary part), and the real and imaginary parts are double values. For example, a complex value 1.2E-3 - 4.56E+7 i is represented as (+1.2E-3, -4.56E+7) in DIF. Matrices are enclosed by brackets, "," is used to separate elements in a row, and ";" is used to separate rows. For example, integer matrices, double matrices, and complex matrices are expressed as [1, 2; 3, 4], [+1.2, -3.4; -0.56e+7, 7.8E-3], and [(1.0, 2.0), (3.0, 4.0); (+1.2, -3.4), (-0.56e+7, 7.8E-3)]. A string value should be double quoted as "string". A boolean value is either True or False. Finally, an array of the aforementioned value types is expressed inside braces, and all elements should be of the same type. For example, we can have an integer array as {1,2,3,4} or double array as {+0.1, -3.6, +1.2E-3, -4.56e+7}. These value types in DIF should be sufficient in most DSP applications. If a certain value type is not supported, it can be handled to some extent by representation through the string type.

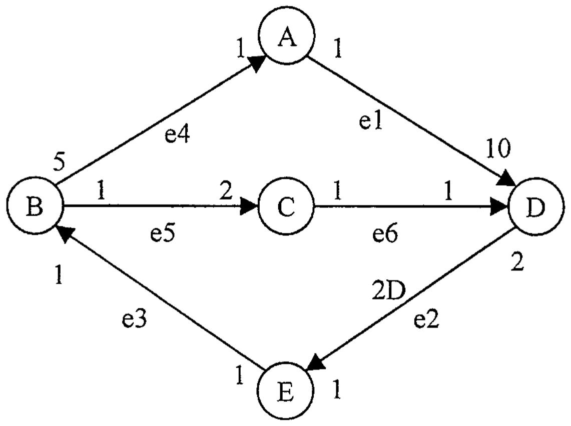
4 Dataflow Models

The DIF language is designed to specify all dataflow models for DSP and streaming related applications. In other words, its syntax and other features should be capable of describing dataflow semantics in all dataflow models of computation relevant to this class of embedded applications. DIF version 0.1 [8] has demonstrated its capability of describing CSDF, SDF, single rate dataflow, and HSDF. DIF version 0.2 improves the feature set to support more complicated dataflow semantics, for example, Turing-complete dataflow such as BDF [6] and meta-modeling techniques such as parameterized dataflow [1]. This section reviews those dataflow models and provides examples to illustrate how to specify them in DIF.

4.1 Synchronous Dataflow

Synchronous dataflow (SDF) [4, 13] is the most popular form of dataflow modeling for DSP design. SDF permits the number of tokens produced and consumed by an actor to be a non-negative integer, which makes it very suitable for modeling *multi-rate* DSP systems. However, SDF also imposes a restriction that production and consumption rates must be fixed and known at compile-time. Therefore, an edge *e* in an SDF graph has three non-negative constant-valued attributes, *prd(e)*, *cns(e)*, and *delay(e)*. The constant restriction on production and consumption rates benefits SDF with the capability of static scheduling, optimization, and predictability but at the cost of limited expressive power, in particular due to lack of support for data-dependent actor interface behavior.

The *dataflowModel* keyword for SDF is *sdf.* The three edge attributes, *prd(e)*, *cns(e)*, and *delay(e)*, are specified in SDF built-in attribute blocks as *production*, *consumption*, and *delay.* Figure 2 illustrates a simple SDF example in DIF.



```
sdf sdfDemo1 {
  topology {
    nodes = A,B,C,D,E;
    edges = e1(A,D), e2(D,E), e3(E,B),
            e4(B,A), e5(B,C), e6(C,D);
  }
  production {
    e1=1; e2=2; e3=1; e4=5; e5=1;e6=1;
  }
  consumption {
    e1=10; e2=1; e3=1; e4=1; e5=2;e6=1;
  }
  delay {
    e2 = 2;
  }
}
```

Figure 2. An SDF example and the corresponding DIF specification.

4.2 Single-rate Dataflow and Homogeneous Synchronous Dataflow

In *single-rate* DSP systems, all actors execute at the same average rate. As a result, the number of tokens produced on an edge when the source node fires is equal to the number of tokens consumed on the same edge when the sink node fires. The *dataflowModel* keyword for single rate dataflow is *SingleRate*. The single rate dataflow model is a special case of SDF, where the production rate and consumption rate of each edge are identical. Because all nodes fire at the same average rate, DIF uses the built-in attribute *transfer* to specify token transfer rates instead of *production* and *consumption* attributes.

In *homogeneous synchronous dataflow* (HSDF), the production rate and consumption rate are restricted to be unity on all edges. HSDF is the simplest widely-used form of dataflow and can be viewed as a restricted case of single-rate dataflow and SDF. The *dataflowModel* keyword for HSDF is *hsdf.* Because of the homogeneous unit transfer rate, specifying *production* and *consumption* attributes is not necessary in HSDF.

Single-rate and HSDF graphs are useful models in scenarios such as uniform execution rate processing, precedence expansion for multi-rate SDF graphs, and multiprocessor scheduling. Algorithms for converting between SDF, single-rate, and HSDF graphs are provided in DIF. Such conversion is illustrated in Figure 3.

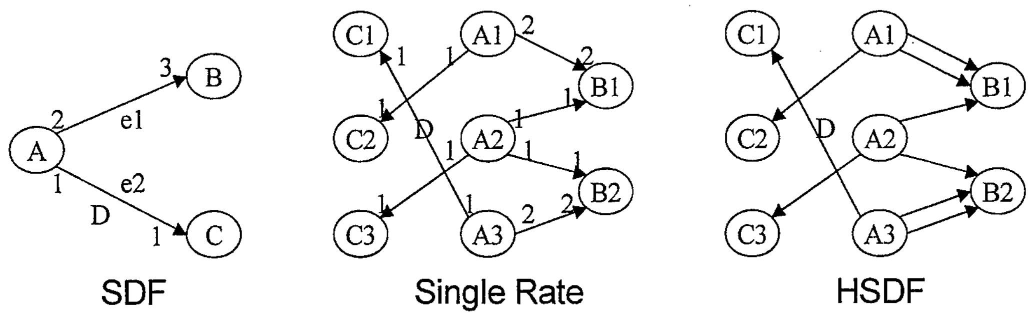
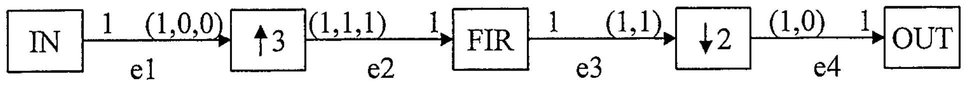


Figure 3. SDF, single rate, and HSDF conversion.

4.3 Cyclo-static Dataflow

In *cyclo-static dataflow* (CSDF) [5], the production rate and consumption rate are allowed to vary as long as the variation forms a fixed and periodic pattern. Explicitly, each actor A in a CSDF graph is associated with a fundamental period $\tau(A) \in Z^{+}$, which specifies the number of phases in one minimal period of the cyclic production / consumption pattern of A. Each time an actor is fired in a period, a different phase is executed. For each incoming edge e of A, *cns(e)* is specified as a $\tau(A)$-tuple $(C_{e,1}, C_{e,2}, ..., C_{e,\tau(A)})$, where each $C_{e,i}$ is a non-negative integer that gives the number of tokens consumed from e by A in the i-th phase of each period of A. Similarly, for each outgoing edge e of A, *prd(e)* is specified as a $\tau(A)$-tuple $(P_{e,1}, P_{e,2}, ..., P_{e,\tau(A)})$, where each $P_{e,i}$ is a non-negative integer that gives the number of tokens produced to e by A in the i-th phase. CSDF offers more flexibility in representing interactions between actors and scheduling, but its expressive power at the level of overall individual actor functionality is the same as SDF.



```
csdf csdfDemo1 {
  topology {
    nodes = IN, UP3, FIR, DOWN2, OUT;
    edges = e1(IN,UP3), e2(UP3,FIR), e3(FIR,DOWN2), e4(DOWN2,OUT);
  }
  production {
    e1=1; e2=[1,1,1]; e3=1; e4=[1,0];
  }
  consumption {
    e1=[1,0,0]; e2=1; e3=[1,1]; e4=1;
  }
}
```

Figure 4. A CSDF example and the corresponding DIF specification.

The *dataflowModel* keyword for CSDF is *csdf*. Built-in attributes *production* and *consumption* are specified as 1-by-$\tau(A)$ integer matrices representing the $\tau(A)$-tuple patterns in one period. Specifically, the DIF specification for a $\tau(A)$-tuple consumption period is specified as: *consumption{ edgeID* = $[C_{e,1}, C_{e,2}, ..., C_{e,\tau(A)}]$*; }*. Figure 4 illustrates an up-sampling and down-sampling example in CSDF.

4.4 Boolean-controlled dataflow

Boolean-controlled dataflow (BDF) [6] is a form of dynamic dataflow for supporting data-dependent DSP computations while still permitting quasi-static scheduling to a certain degree. BDF is Turing-complete [6]. Quasi-static scheduling refers to a form of scheduling in which a significant proportion of scheduling decisions is made at compile-time through analysis of static properties in the application model. By including BDF, DIF improves its ability to explore Turing-complete semantics and incorporates detailed support for an important, fully expressive model.

In dynamic dataflow modeling, a *dynamic actor* produces or consumes certain numbers of tokens depending on the incoming data values during each firing. In BDF, the number of tokens produced or consumed by a dynamic actor is restricted to be a two-valued function of the value of certain "control tokens." In other words, the number of tokens that a boolean-controlled actor A produces to an edge e_o or consumes from an edge e_i during each firing is determined by *TRUE* or *FALSE* values of the control token consumed by A at that iteration, where $e_o \in out(A)$ or $e_i \in in(A)$. BDF also imposes a restriction that a boolean controlled actor can only consume one control token during each firing. The following two equations describe the boolean-controlled production and consumption rates in BDF.

$$prd(e_o) \text{ at i-th iteration} = \begin{cases} \text{prod rate1, if control token consumed by } A \text{ at i-th iteration is } TRUE \\ \text{prod rate2, if control token consumed by } A \text{ at i-th iteration is } FALSE \end{cases}$$

$$cns(e_i) \text{ at i-th iteration} = \begin{cases} \text{cons rate1, if control token consumed by } A \text{ at i-th iteration is } TRUE \\ \text{cons rate2, if control token consumed by } A \text{ at i-th iteration is } FALSE \end{cases}$$

In addition to boolean-controlled dynamic actors, other actors are required to be *regular.* A regular actor produces and consumes fixed and known numbers of tokens at compile-time; it is similar to an SDF actor.

The *dataflowModel* keyword for BDF is *bdf.* Built-in attributes *production* and *consumption* can be used to specify both fixed and boolean-controlled production and consumption rates. For a fixed rate, the syntax is the same as SDF; for a boolean-controlled rate, a 1-by-2 integer matrix is utilized to specify two boolean-controlled values. The first element is the rate when the control token is *TRUE,* and the second element is the rate when the control token is *FALSE.* Specifically, the following syntax shows the DIF specification for a boolean-controlled production rate.

```
production { edgeID = [trueValue, falseValue]; }
```

BDF introduces two boolean-controlled actors, *SWITCH* and *SELECT.* The SWITCH actor consumes one token from its incoming edge and copies that token to either a “true” outgoing edge or a “false” outgoing edge according to the value of a control token. The SELECT actor consumes one token from a either “true” incoming edge or a “false” incoming edge according to the value of a control token and copies that token to the outgoing edge. Figure 5 illustrates a BDF example implementing an *if-else* statement.

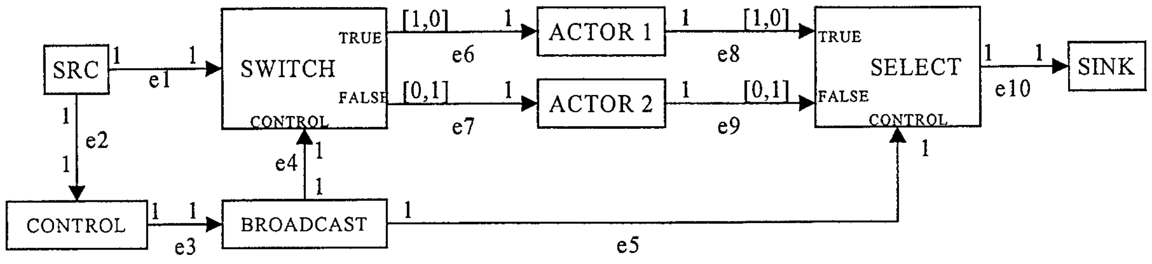
4.5 Parameterized Synchronous Dataflow

Parameterized dataflow modeling differs from other fundamental dataflow modeling techniques such as SDF, CSDF, in that it is a *meta-modeling* technique. Parameterized dataflow can be applied to any underlying “base” dataflow model that has a well-defined notion of a graph iteration. Applying parameterized dataflow in this way augments the base model with powerful capabilities for dynamic reconfiguration and quasi-static scheduling through parameterized looped schedules [1]. Combining parameterized dataflow with synchronous dataflow forms *parameterized synchronous dataflow* (PSDF), a dynamic dataflow model that has been investigated in depth and shown to have useful properties [1].

A PSDF actor A is characterized by a set of parameters, *params(A)*, that can control the actor’s functionality as well as the actor’s dataflow behavior such as production rates and consumption rates. A configuration of a PSDF actor, $config_A$, is determined by assigning values to the parameters of A. Each parameter of an actor is either assigned a value or left unspecified. These statically unspecified parameters are assigned values at run time, thus dynamically modifying the actor’s functionality.

A PSDF graph G is an ordered pair (V, E) and all statically unspecified actor parameters in G propagate “upwards” as parameters of the PSDF graph G, which are denoted as *params(G)*. A DSP application is usually modeled in PSDF through a *PSDF specification,* which is also called a *PSDF subsystem.* A PSDF subsystem Φ consists of three PSDF graphs, the *init graph* Φ_i, the *subinit graph* Φ_s, and the *body graph* Φ_b. The body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring parameters of the body graph. Moreover, PSDF employs a hierarchical modeling structure by allowing a PSDF subsystem Φ to be embedded in a “parent” PSDF graph G and abstracted as a hierarchical PSDF actor H, where $\Phi = subsystem(H)$.

The init graph Φ_i does not take part in the dataflow and all the parameters of Φ_i are left unspecified. The subinit graph Φ_s may only accept dataflow inputs at its interface input ports and each parameter of Φ_s is configured either by an interface output port of Φ_i, is set by an interface input port of Φ, or is left unspecified. The interface output ports of Φ_i and Φ_s are reserved exclu-



```
if CONTROL outputs TRUE token
   fire ACTOR1;
else
   fire ACTOR2;
end
```

```
bdf bdfDemo1 {
  topology {
    nodes = SRC, SWITCH, SELECT, SINK, CONTROL, BROADCAST, ACTOR1, ACTOR2;
    edges = e1(SRC,SWITCH), e2(SRC,CONTROL), e3(CONTROL,BROADCAST),
            e4(BROADCAST,SWITCH), e5(BROADCAST,SELECT), e6(SWITCH,ACTOR1),
            e7(SWITCH,ACTOR2), e8(ACTOR1,SELECT), e9(ACTOR2,SELECT),
            e10(SELECT,SINK);
  }
  production {
    e1=1; e2=1; e3=1; e4=1; e5=1; e6=[1,0]; e7=[0,1]; e8=1; e9=1; e10=1;
  }
  consumption {
    e1=1; e2=1; e3=1; e4=1; e5=1; e6=1; e7=1; e8=[1,0]; e9=[0,1]; e10=1;
  }
  actor SWITCH {
    computation; = "dif.bdf.SWITCH";
    control : CONTROL = e4;
    input : INPUT = e1;
    true : TRUEOUTPUT = e6;
    false : FALSEOUTPUT = e7;
  }
  actor SELECT {
    computation; = "dif.bdf.SELECT";
    control : CONTROL = e5;
    output : OUTPUT = e10;
    true : TRUEINPUT = e8;
    false : FALSEINPUT = e9;
  }
```

Figure 5. A BDF example, the corresponding pseudocode, and DIF specification.

sively for configuring parameter values. The body graph Φ_b usually takes on the major role in dataflow processing and all of its dynamic parameters are configured by the interface output ports of Φ_i and Φ_s. All unspecified parameters of Φ_i and Φ_s propagate "upwards" as the subsystem parameters of Φ, which are denoted as *params(Φ)* and are configured by the init and subinit graphs of hierarchically higher level subsystems. This mechanism of parameter configuration is referred as *initflow.*

In order to maintain a valuable level of predictability and efficient quasi-static scheduling, PSDF requires that the interface dataflow of a subsystem must remain unchanged throughout any given iteration of its hierarchical parent subsystem. Therefore, parameters that determine the interface dataflow can only be configured by output ports of the init graph Φ_i, and Φ_i is only invoked once at the beginning of each invocation of the supergraph. As a result, the parent has a consistent view of module interfaces throughout any iteration. On the other hand, parameter reconfiguration that does not change interface behavior of subsystem is permitted to occur across iterations of the subsystem rather than the parent subsystem. The subinit graph Φ_s performs this reconfiguration activity and is invoked each time before an invocation of the body graph Φ_b. This gives a subsystem a consistent view of its components' configurations throughout any given iteration and provides configurability across iterations.

Specifying such complicated meta-modeling techniques in a fully general way (so that they can operate with other models a maximally flexible way) in DIF is a challenging task. DIF separates PSDF graphs and PSDF subsystems into two modeling blocks, and the corresponding *dataflowModel* keywords for them are *psdf* and *psdfSubsystem*, respectively. Parameterization is a main feature of DIF with the parameter block and this feature is very suitable in specifying PSDF. Configurable actor attributes and non-static dataflow modeling attributes such as production rates and consumption rates are parameterized by pre-defined parameters. Unspecified parameters are defined without providing their values in the parameter block. *Upward parameters* of a PSDF subsystem can be specified in the refinement block of its supergraph. For hierarchical modeling structures in PSDF, e.g.,., $\Phi = subsystem(H)$, the DIF hierarchy concepts described in Section 2.2 can fully represent the associated functionality and the DIF refinement block is used to specify them.

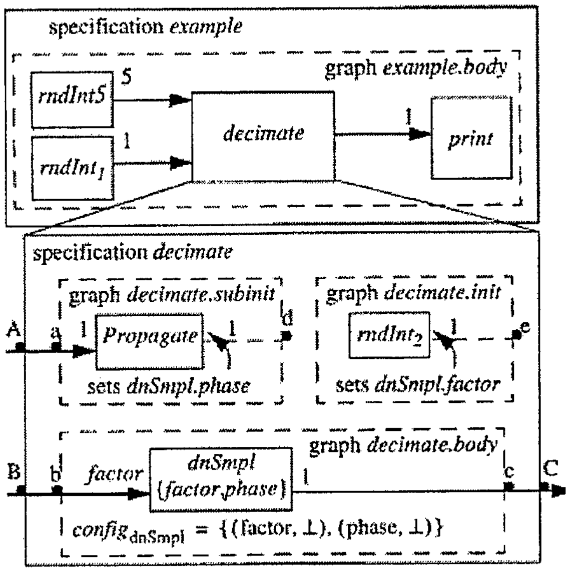
DIF interprets a PSDF subsystem as a special intermediate graph that consists of three subgraphs, Φ_i, Φ_s, and Φ_b. In DIF specification, a PSDF subsystem cannot have the topology block because the three subgraphs, init, subinit, and body (Φ_i, Φ_s, and Φ_b) are built-in and there is no edge connection in any PSDF subsystem. The parameter reconfigurations across init, subinit, and body graphs are specified in the built-in attribute block called *paramConfig* with the following syntax.

```
paramConfig {
   subinitGraphID.paramID = initGraphID.outputPortID;
   bodyGraphID.paramID = initGraphID.outputPortID;
   bodyGraphID.paramID = subinitGraphID.outputPortID;
}
```

Figure 6 illustrates a PSDF example in [1] and the corresponding DIF specification.

4.6 Binary Cyclo-static Dataflow

Binary CSDF (BCSDF) is a restricted form of CSDF such that the production and consumption rates are constrained to be binary vectors. In other words, elements of the BCSDF production and consumption vectors are either 0 or 1. BCSDF graphs arise naturally, for example, when convert-



```
psdf decimateSubinit {
  topology { nodes = Propagate; }
  interface {
    inputs = a:Propagate;
    outputs = d:Propagate;
  }
  consumption { a = 1; }
  production { d = 1; }
}

psdf decimateInit {
  topology { nodes = rndInt2; }
  interface { outputs = e:rndInt2; }
  production { e = 1; }
}

psdf decimateBody {
  topology { nodes = dnSmpl; }
  interface {
    inputs = b:dnSmpl;
    outputs = c:dnSmpl;
  }
  parameter {
    factor;
    phase;
  }
  consumption { b = factor; }
  production { c = 1; }
}
```

```
psdfSubsystem decimate {
  interface {
    inputs = A:subinit, B:body;
    outputs = C:body;
  }
  refinement { decimateInit = init; }
  refinement {
    decimateSubinit = subinit;
    a : A;
  }
  refinement {
    decimateBody = body;
    b : B;
    c : C;
  }
  paramConfig {
    decimateBody.factor =
        decimateInit.e;
    decimateBody.phase =
        decimateSubinit.d;
  }
}

psdf exampleBody {
  topology {
    nodes = rndInt5, rndInt1,
            decimate, print;
    edges = e1(rndInt5, decimate),
            e2(rndInt1, decimate),
            e3(decimate, print);
  }
  refinement {
    decimate = decimate;
    A : e1; B : e2; C : e3;
  }
  production {
    e1 = 5; e2 = 1;
  }
  consumption { e3 = 1; }
}

psdfSubsystem example {
  refinement { decimate = body; }
}
```

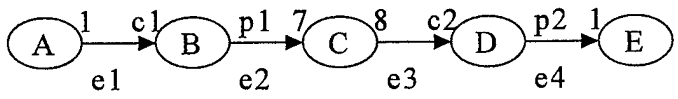
Figure 6. A PSDF example and the corresponding DIF specification.

ing certain process networks to dataflow and when modeling many dataflow-based hardware implementations.

The DIF specification format for BCSDF is as the same as CSDF. In some BCSDF representations, the numbers of phases can be very large. Therefore, the BCSDF intermediate representation utilizes an efficient data structure to store the production and consumption rates as bit vectors.

4.7 Interval-Rate Locally-static Dataflow

Interval-Rate Locally-static Dataflow (ILDF) [17] is proposed to analyze dataflow graphs whose component data rates are not known precisely at compile time. In ILDF graphs, the production and consumption rates remain constant throughout execution (locally-static), but only the minimum and maximum values (interval-rate) of these constants are given. DIF is capable of representing ILDF graphs by parameterizing the ILDF production and consumption rates and specifying the intervals of those parameters, which is described in Section 3.6. Figure 7 illustrates an ILDF example and the corresponding DIF specification.



```
ildf ildfDemo1 {
  topology {
    nodes = A, B, C, D, E;
    edges = e1(A,B), e2(B,C), e3(C,D), e4(D,E);
  }
  parameter {
    c1 : [3,7];
    c2 : [3,7];
    p1 : [2,10];
    p2 : [2,10];
  }
  production {
    e1 = 1; e2 = p1; e3 = 8; e4 = p2;
  }
  consumption {
    e1 = c1; e2 = 7; e3 = c2; e4 = 1;
  }
}
```

Figure 7. An ILDF example and the corresponding DIF specification.

5 The DIF Package

The DIF package is a Java software package developed along with the DIF language. In general, it consists of three major parts: the DIF front-end, the DIF representation, and the implementations of dataflow-based analysis, scheduling, and optimization algorithms. This section introduces the major parts of the DIF package and describes the relationship of the DIF package to theoretical dataflow models, dataflow-based DSP design tools, and underlying embedded processing platforms.

5.1 The DIF Representation

For each supported dataflow model, the DIF package provides an extensible set of data structures (object-oriented Java classes) for representing and manipulating dataflow graphs in the model. This graph-theoretic intermediate representation for the dataflow model is usually referred to as the *DIF representation.*

The DIFGraph is the most general graph class in the DIF package. It represents the basic dataflow graph structure among all dataflow models and provides methods that are common to all models for manipulating graphs. For a more specialized dataflow model, development can proceed naturally by extending the general DIFGraph class (or some suitable subclass) and overriding and adding new methods to perform more specialized functions.

Figure 8 presents the class hierarchy of graph classes in the DIF package. The DIFGraph is extended from the DirectedGraph class, and in turn, from the Graph class. The DirectedGraph and Graph classes are used from the Ptolemy II [10] ptolemy.graph package, which is developed in collaboration between members of the DIF and Ptolemy projects, and provides data structures and methods for manipulating generic graphs. The dataflow models CSDF, SDF, single rate dataflow, and HSDF are related in such a way such that each succeeding model among these four is a special case of the preceding model. Accordingly, CSDFGraph, SDFGraph, SingleRateGraph, and HSDFGraph form a class hierarchy in the DIF package such that each succeeding graph class inherits from the more general one that precedes it (see Figure 8).

In addition to the aforementioned fundamental dataflow graph classes, the DIF package also provides the Turing-complete BDFGraph, the meta-modeling PSDFGraph, and BCSDFGraph for the newly introduced BCSDF model. Furthermore, a variety of other dataflow models are being explored for inclusion in DIF.

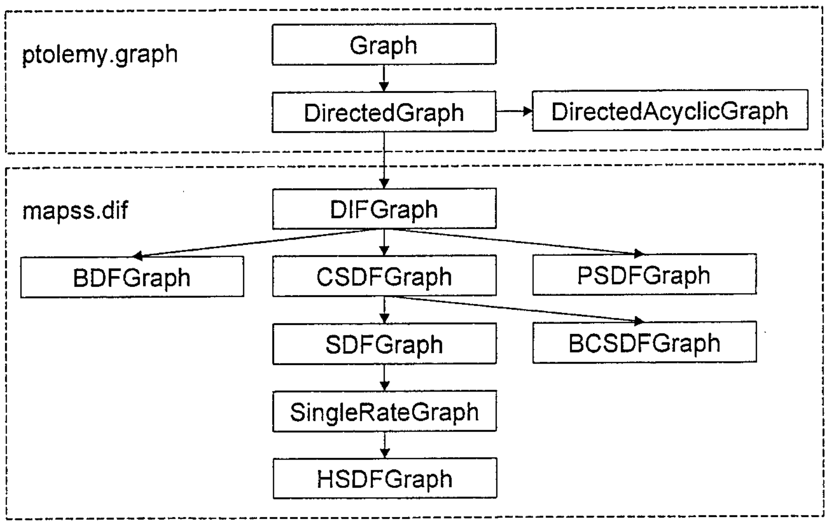


Figure 8. The class hierarchy of graphs in the DIF package.

5.2 The DIF Front-end

Although the DIF language is able to specify all dataflow models, in reality, the DIF representation is the actual format for realizing dataflow graphs and for performing analysis, scheduling, and optimization. Thus, automatic conversion between DIF specifications (.dif files) and DIF representations (graph instances) is the most fundamental feature of the DIF package. The DIF front-end tool automates this conversion and provides users an integrated set of programming interfaces to construct DIF representations from specifications and to generate DIF specifications from intermediate representations.

The DIF front-end consists of a Reader class, a set of language parsers (LanguageAnalysis classes), a Writer class, and a set of graph writer classes. The language parsers are implemented using a Java-based compiler compiler called SableCC [7]. The flexible structure and Java integration of the SableCC compiler enables easy extensibility for parsing different dataflow graph types.

Figure 9 illustrates how the DIF front-end constructs the corresponding DIF representation (graph class) according to the given DIF specification. The Reader class invokes the corresponding language analysis class (DIF language parser) based on the model keyword specified in the DIF specification. Then, the language analysis class constructs a graph instance according to the dataflow semantics specified in the DIF specification.

On the other hand, Figure 10 illustrates how the DIF front-end generates a DIF specification according to a DIF representation. The Writer class invokes the corresponding graph writer class based on the type of the given graph instance. After that, the graph writer class generates the DIF specification by tracing elements and attributes of the graph instance.

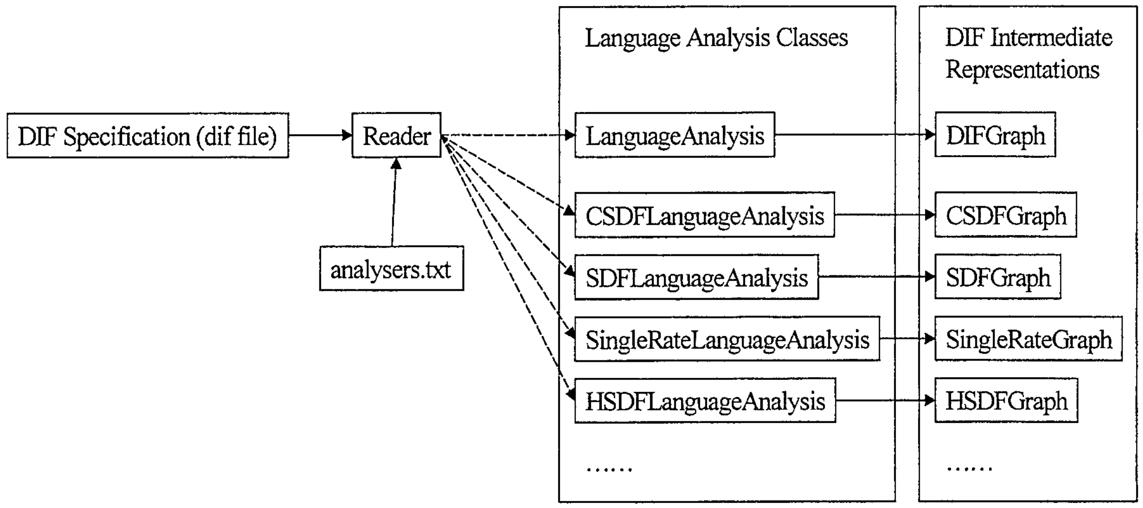


Figure 9. The DIF Front-end: from DIF specification to DIF representation.

In the DIF package, the language analysis classes (language parsers) are used for parsing most parts of the DIF language, except for built-in attribute blocks. Similarly, the graph writer classes are used for writing out most parts of the dataflow semantics, except for the methods handling the built-in attributes. Therefore, all specialized dataflow language analysis classes are extended from the LanguageAnalysis class that constructs the most general DIFGraph. Likewise, all specialized graph writer classes are extended from the DIFWriter class, which writes out the dataflow semantics of DIFGraph instances.

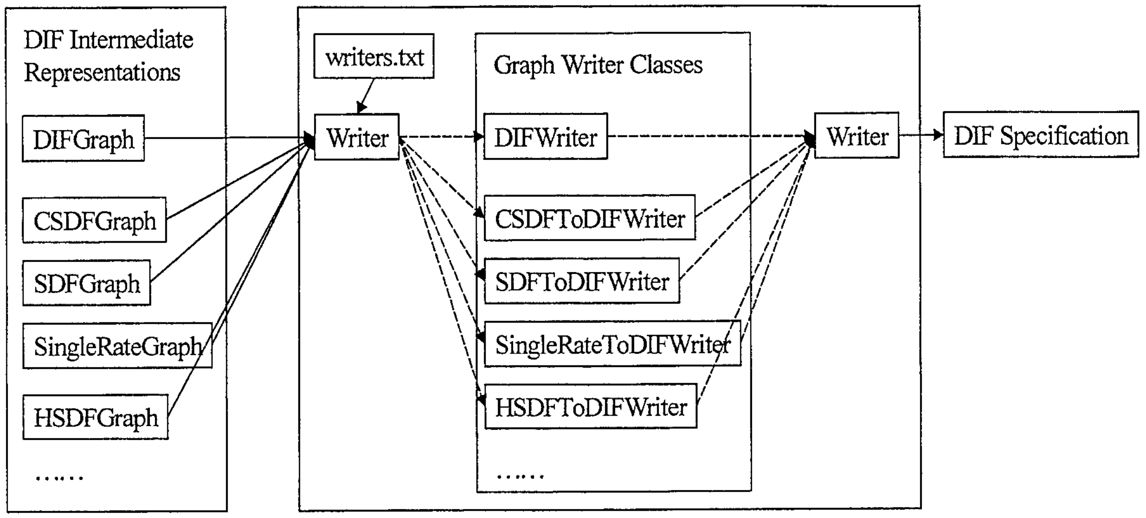


Figure 10. The DIF front-end: from DIF representation to DIF specification.

5.3 Dataflow-based Analysis, Scheduling, and Optimization Algorithms

For supported dataflow models, the DIF package provides not only graph-theoretic intermediate representations but also efficient implementations of various useful analysis, scheduling, and optimization algorithms that operate on the representations. Algorithms currently available in the DIF package are based primarily on well-developed algorithms such as iteration period computation, consistency validation, buffer minimization, and loop scheduling. By building on the DIF representations and existing algorithm implementations, and invoking the built-in algorithms as needed, emerging techniques and other new algorithm implementations can conveniently be developed and implemented in the DIF package.

The dataflow-based algorithms in the DIF package provide designers an efficient interface to analyze and optimize DSP applications. It is also worthwhile to integrate DSP design tools with the DIF package and then utilize the powerful scheduling and optimization features of the DIF package.

5.4 Methodology of using DIF

Figure 11 illustrates the conceptual architecture of DIF and the relationships among abstract dataflow models, dataflow-based DSP design tools, DIF specifications, and the DIF package. First of all, the dataflow model block in this diagram presents the dataflow models currently supported in DIF. Based on the DIF language introduced in Section 3, application models using these dataflow models can be specified as DIF specifications, which are described in Section 4.

The block for dataflow-based design tools represents currently available and other previously developed DSP design tools. These tools usually provide a block-diagram-based graphical design environment, a set of libraries consisting of useful modules, and a programming interface for designing modules. As long as the DSP system modeling capability in a design tool is based on dataflow principles, the DIF language is able to capture the associated dataflow semantics and related modeling information of DSP applications in the tool and represent them in the form of DIF specifications.

The DIF package realizes the abstract dataflow structure of DSP application models through the DIF representation. With the DIF front-end tool, the DIF representation can be constructed automatically based on the given DIF specification. After that, dataflow-based analysis, scheduling, and optimization techniques can be applied on the DIF representation.

Figure 12 illustrates the implementation and end-user viewpoints of the DIF architecture. DIF supports as the core a layered design methodology covering dataflow models, the DIF language and DIF specifications, the DIF package, dataflow-based DSP design tools, and the underlying hardware and software platforms targeted by these tools.

The dataflow models layer represents the dataflow models currently integrated in the DIF package. These models can be further categorized into static dataflow models such as SDF and CSDF; dynamic dataflow models such as the Turing-complete BDF model; and meta-modeling techniques such as parameterized dataflow, which provides the dynamic reconfiguration capability of PSDF. Using the DIF language, application behaviors compatible with these dataflow modeling techniques can be specified in a streamlined manner as specialized DIF specifications.

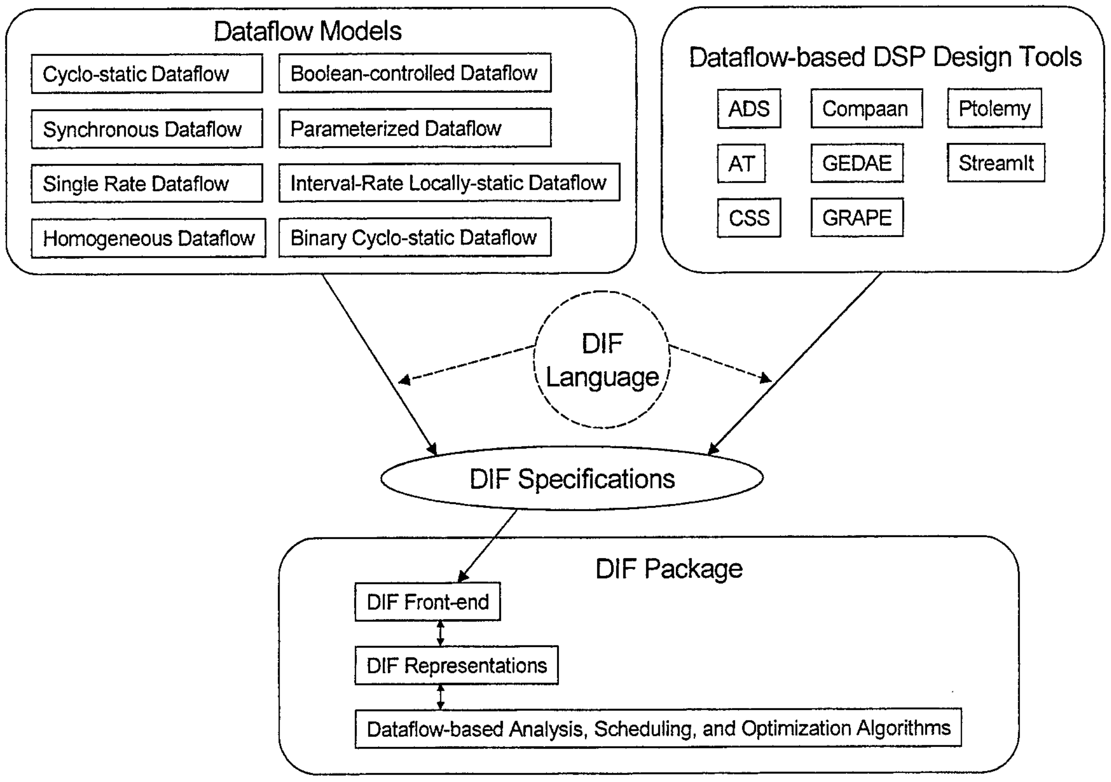


Figure 11. The relationships among dataflow models, design tools, the DIF language, DIF specifications, and the DIF package.

The primary dataflow-based DSP design tools that we have been experimenting with in our development of DIF so far are the SDF domain of Ptolemy II, developed at UC Berkeley, and the Autocoding Toolset developed by MCCI. However, DIF is in no way designed to be specific to these tools; they are used only as a starting point for experimenting with DIF in conjunction with sophisticated academic and industrial DSP design tools, respectively. Tools such as these form a layer in our proposed DIF-based design methodology. Ptolemy II is a Java-based design environment and utilizes the Modeling Markup Language (MoML) as its textual format for specification and interchange. Ptolemy II provides multiple models of computation and a large set of libraries consisting of actors for various application domains. On the other hand, the MCCI Autocoding Toolset is based on the Processing Graph Method (PGM) semantics and uses Signal Processing Graph Notation (SPGN) as its specification format. It also provides an efficient library consisting of domain primitives for DSP computations and is able to synthesize software implementations for certain high-performance platforms.

The hardware / software embedded systems layer gives examples of current embedded processing platforms supported by Ptolemy II and the Autocoding Toolset, and this layer generally represents all embedded platforms that are supported by dataflow-based DSP design tools. Ptolemy II can generate executable Java code running on the Java VM. On the other hand, the Autocoding toolset is able to generate executable C code for Mercury DSPs and Ada for the Virtual Design Machine (VDM) [16]. In addition, we are examining the requirements and implications of DIF-based support for other tools that have the ability to map dataflow models to efficient hardware / software implementations

The DIF package acts as an intermediate layer between abstract dataflow models and different practical implementations. It takes the responsibility of realizing dataflow graphs and performing dataflow-based algorithms. DIF exporting and importing tools automate the process of exporting DSP applications from design tools to DIF specifications and importing them back to design tools. Automating the exporting and importing processes between DIF and design tools provides the DSP design industry a useful front-end to use DIF and the DIF package. In the next section, we will describe issues involved in such automation, and our approaches to addressing these issues.

6 Exporting and Importing DIF

The DIF language is capable of specifying dataflow semantics of DSP applications in any dataflow-based design tool. When integrating features of DIF with a DSP design tool, incorporating capabilities to translate between the design tool's specification format and DIF specifications or DIF representations is usually an essential first step. In DIF terminology, *exporting* means translating a DSP application from a design tool's specification format to DIF (either to the DIF language or directly to the appropriate form of DIF representation). On the other hand, *importing* means translating a DIF specification to a design tool's specification format or converting a DIF representation a to design tool's internal representation format. Figure 13 illustrates the exporting and importing mechanisms between DIF and design tools.

When exporting, parsing design tools' specification formats and then directly formulating the corresponding DIF specifications is usually not an efficient way. In contrast, DIF provides a complete set of classes for representing dataflow graphs in a well-designed, object-oriented realization. Hence, instead of parsing and directly formulating equivalent DIF language code, mapping design tools' graphical representations to DIF representations and then converting to DIF specifications using representation-to-specification translation capabilities already built in to DIF is typically much easier and more efficient.

However, depending on the particular design tool involved, it still may be a somewhat involved task to automate the exporting and importing processes. First of all, graph topologies

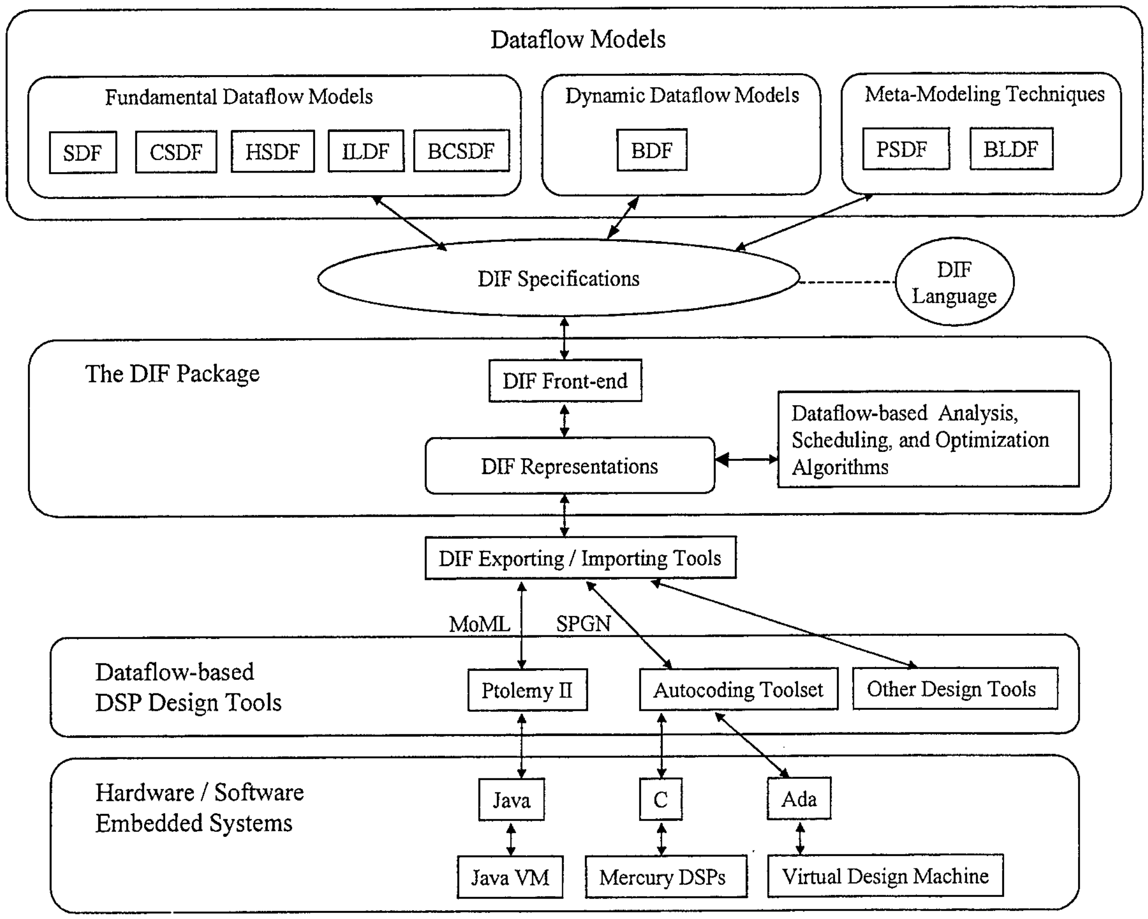


Figure 12. The role of DIF in DSP system design..

and hierarchical structures of DSP applications must be captured in order to completely represent their dataflow semantics. Furthermore, actors' computations, parameters, and connections must also be specified for preserving application functionality completely. In the following subsections, explain more about these issues and describe our approaches to addressing them. For illustration, we also demonstrate DIF-Ptolemy exporting and importing capabilities that we have developed.

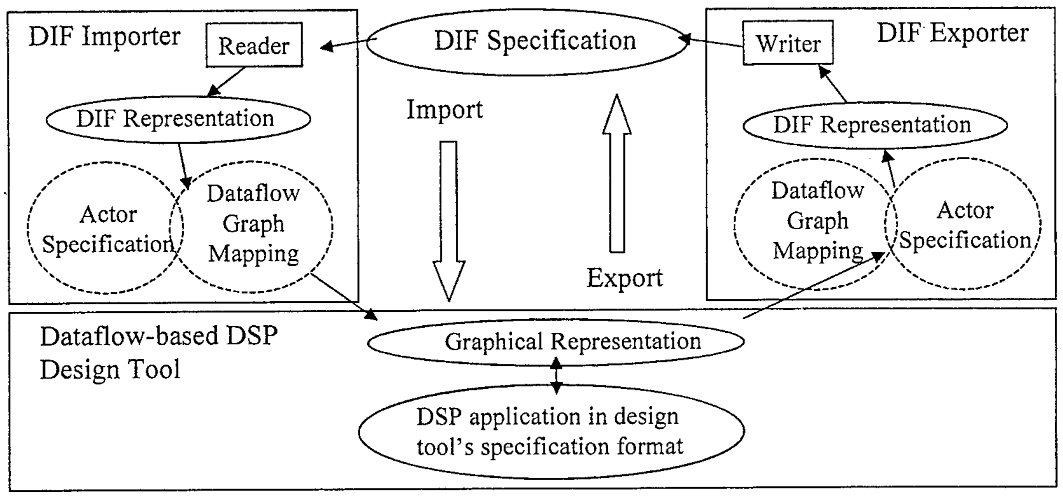


Figure 13. Exporting and Importing Mechanism

6.1 Mapping Dataflow Graphs

Dataflow-based DSP design tools usually have their own representations for nodes, edges, hierarchies, etc. Moreover, they often use more specific components instead of just the abstract components found in formal dataflow representations. Implementation issues involved in converting the graphical representations of design tools to the formal dataflow representations used in DIF are categorized as *dataflow graph mapping* issues.

Let's take exporting Ptolemy II to DIF as an example to explain problems in dataflow graph mapping. Ptolemy II has the *AtomicActor* class for representing DSP computations (associated with primitive dataflow nodes) and the *CompositeActor* class for representing subgraphs. It uses the *Relation* class instead of edges to connect actors. Each actor has multiple *IOPorts* and those *IOPorts* are connection points for *Relations*. A *Relation* can have a single source but fork to multiple destinations. Regular *IOPorts* can accept only one *Relation* but Ptolemy II also allows *multiport IOPorts* that can accept multiple *Relations*. Clearly, problems arise when mapping Ptolemy II graphical representations to DIF representations. First, based on the formal definition of nodes in dataflow models, they do not have ports to distinguish interfaces. Second, edges in formal dataflow graphs cannot support multiple destinations in contrast to Ptolemy II *Relations*. Third, the *multiport* property in Ptolemy II does not match with formal dataflow semantics, and even an interface port of a hierarchy defined in Section 2.2 can only connect to one outer edge or port.

Although implementation problems in dataflow graph mapping are tool-specific, exporting without losing any essential modeling information is still feasible due to the broad range of modeling capabilities offered through the features in DIF. First, the DIF language is capable of describing dataflow semantics regardless of the particular design tool used to enter an application model as long as the tool is dataflow-based. Second, DIF representations can fully realize the dataflow graphs specified by the DIF language. Based on these two properties, our general approach comprehensively traverse graphical representations in a design tool and then map the modeling components encountered to equivalent components or groups of components available for DIF representations. After that, our DIF front-end tool can write the DIF representations into textual DIF specifications.

A brief description follows of the algorithm developed for mapping Ptolemy's graphical representations to DIF representations. First, *AtomicActor*s are represented by nodes and *CompositeActor*s are represented by hierarchies. Single-source-single-destination *Relation*s are represented by edges. For a multiple-destination *Relation,* a fork actor (which is described in Section 6.3) and several edges are used to represent it without losing any dataflow properties. A Ptolemy II actor's *IOPort*s and the corresponding connections are specified as actor attributes. Even for a *multiport IOPort,* multiple connections can still be listed as an actor attribute.

6.2 Specifying Actors

In dataflow analysis [14], a node may be viewed as a functional unit associated with a weight that consumes/produces certain numbers of tokens when executing. Usually, dataflow-based analysis and scheduling techniques are based on production rates, consumption rates, edge delays, and various node weight information and other edge weight information (e.g., node execution times or execution time distributions, and the interprocessor communication cost associated with an edge if its source and sink are mapped to different processors in a multiprocessor target). Thus, the detailed computation performed by a node is irrelevant to many dataflow-based analyses. However, the computation (such as an FFT operation) and attributes (such as the order of the FFT) associated with a node is essential during implementation. To avoid confusion between the viewpoints of nodes in dataflow analyses versus in hardware/software implementations, we henceforth use the term *node* for the former context, and we use the term *actor* to refer to a node with specified computation and other implementation-related attributes (for the latter context).

Specifying an actor's computation as well as all necessary operational information is referred to as *actor specification.* It is an important issue in exporting and importing between DIF and design tools as well as in porting DSP applications across tools because every actor's functionality must be preserved. The *actor block* is newly added to DIF language version 0.2 for actor specification. The DIF language syntax for the actor block is described in Section 3.10. Note that for most dataflow-based analysis and scheduling techniques, the DIF language syntax without the actor block is sufficient.

To illustrate actor specification, we take the FFT operations in Ptolemy II and in the Autocoding Toolset as examples. In Ptolemy II, actors are implemented in Java and invoked through their classpath. The FFT actor definition in Ptolemy II is thus referred to as

'ptolemy.domains.sdf.lib.FFT.

In the Autocoding Toolset, actors are called *domain primitives*, and each domain primitive is referred to by its library identifier. The FFT domain primitive in the Autocoding Toolset is referred to as *D_FFT.*

In exporting Ptolemy II to DIF, an actor's parameters and *IOPort-Relation* connections are specified as actor attributes. The built-in attributes PARAMETER, INPUT, and OUTPUT in DIF indicate the parameters and interface connections of an actor. A full DIF actor block for a Ptolemy FFT actor is presented below:

```
actor nodeID {
   computation = "ptolemy.domains.sdf.lib.FFT;
   order : PARAMETER = integerValue or integerParameterID;
   input : INPUT = incomingEdgeID;
   output : OUTPUT = outgoingEdgeID;
}
```

A Ptolemy FFT actor has a parameter *order* and two IOPorts, *input* and *output*. Therefore, in the corresponding DIF actor specification, attribute *order* (with attribute type PARAMETER) specifies the FFT order. In addition, attributes *input* (with attribute type INPUT) and *output* (with attribute type OUTPUT) specify the incomingEdgeID and outgoingEdgeID connecting to the corresponding IOPorts.

In the Autocoding Toolset, input/output connections and function configuration parameters of a domain primitive are all viewed as parameters. In the D_FFT domain primitive, parameter *X* specifies its input, parameter *Y* specifies its output, and parameter *N* specifies its length. In this case, the components of actor-specific information are all of the same tool-specific class (parameter), so the *attributeType* field in the DIF specification can simply be ignored. There is no loss of of information in leaving them out. The corresponding DIF specification for the D_FFT domain primitive is presented below.

```
actor nodeID {
   computation = "D_FFT";
   N = integerValue or integerParameterID;
   X = incomingEdgeID;
   Y = outgoingEdgeID;
}
```

6.3 The Fork Actor

The fork actor is introduced in DIF as a special built-in actor. It can have one and only one incoming edge and multiple outgoing edges. Conceptually, when firing, the fork actor consumes a token from its incoming edge and duplicates the same token on each of its outgoing edges. We say "conceptually" here because in an actual implementation of the fork actor, it may be desirable to achieve the same effect through careful arrangement and manipulation of the relevant buffers. The fork actor is widely used in dataflow. For example, if a stream of data tokens is required to be "broadcast" to multiple destinations, the fork actor can be used for this purpose. The built-in DIF computation associated with the fork actor is called *dif.fork*.

In dataflow theory, an edge is a data path from a source node to a sink node. It cannot be associated with multiple sink nodes. But the *Relation* in Ptolemy II can have multiple destinations. In order to export Ptolemy's graphical representations to DIF representations, the graph mapping algorithm must be able to take care of this structural difference. By using a fork actor, an edge connecting to the input of the fork actor, and multiple edges connecting from the fork actor to all sink nodes, we can represent the Ptolemy *Relation* and preserve the same dataflow semantics while using the formal dataflow representations in DIF.

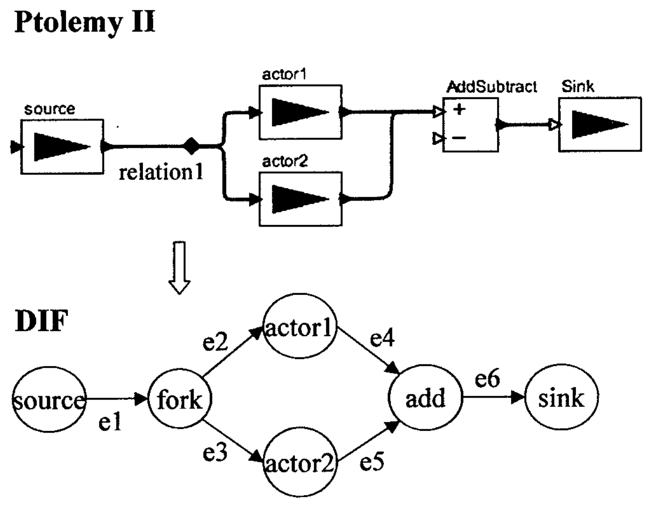
Figure 14 illustrates how the fork actor and actor specification solve the Ptolemy *Relation* and Ptolemy *multiport* problems.

6.4 Exporting and Importing Tools

In order to provide a front-end for a dataflow-based design tool to cooperate with DIF and to use the DIF package, automating the exporting and importing processes for the design tool is the most important feature. Figure 13 illustrates our proposed exporting and importing mechanism. First, a dataflow graph mapping algorithm must be properly designed for the specific design tool that is being used. Then a DIF exporter is implemented for that design tool based on the graph mapping algorithm. It must be able to convert the graphical representation format in that tool to a corresponding DIF representation. Actor specification is also required to preserve the full functionality of actors. By applying the DIF front-end, the DIF exporter can translate the DIF representation to a corresponding DIF specification and complete the exporting process.

Similarly, by using the DIF front-end, the DIF importer can read the DIF specification and generate the DIF representation. Then, based on a "reverse graph mapping algorithm" and actor specification, the DIF importer is able to construct the graphical representation in the design tool while preserving the same functionality of the original DSP application.

The DIF exporter and DIF importer for Ptolemy II are implemented according to the exporting and importing mechanism described above. With these software components, a DSP application in Ptolemy II can be exported to a DIF specification and then be imported back to a Ptolemy MoML specification with all functionality preserved. Such an equivalent result from round-trip translation validates the correctness of the implemented strategies and general methods in DIF for dataflow graph mapping and actor specification.



```
dif graph1 {
  topology {
    nodes = source, fork, actor1, actor2, add, sink;
    edges = e1 (source, fork), e2 (fork, actor1), e3 (fork, actor2),
             e4 (actor1, add), e5 (actor2, add), e6 (add, sink);
  }
  actor fork {computation = "dif.fork";}
  actor add {
    computation = "dif.actor.lib.AddSubtract";
    plus : INPUT = e4, e5;
    output : OUTPUT = e6;
  }
}
```

Figure 14. Mapping the Ptolemy II graphical representation to a DIF representation and the corresponding DIF specification.

7 Porting DSP Applications

DIF is proposed to be a standard language for specifying dataflow graphs in all well-defined dataflow models. One of the original goals was to transfer information associated with DSP applications across different dataflow-based design tools. This goal was demonstrated in the first version of DIF [8, 12].

In the development of DIF version 0.2, we have further explored this direction and developed a new and significantly improved approach for porting dataflow-based DSP applications across design tools. The objective of this porting mechanism is to provide, with a high degree of automation, a solution such that an application constructed in one design tool can be ported to another design tool with enough details preserved throughout the translation to ensure executability on the associated set of target embedded processing platforms. Because different design tools support different sets of underlying embedded processing platforms, porting DSP applications across design tools is effectively equivalent to porting them across those underlying platforms. Thus, the proposed DIF porting mechanism not only facilitates technology transfer at the level of application models, but also provides portability across target platforms.

In this section, we introduce the porting mechanism in detail. In the next section, we demonstrate that this mechanism is a feasible solution through an example of a synthetic aperture radar (SAR) benchmark application that is transferred between the MCCI Autocoding Toolset and Ptolemy II. These tools are significantly different in nature and the ability to automatically port an important application like SAR across them is a useful demonstration of the DIF porting mechanism.

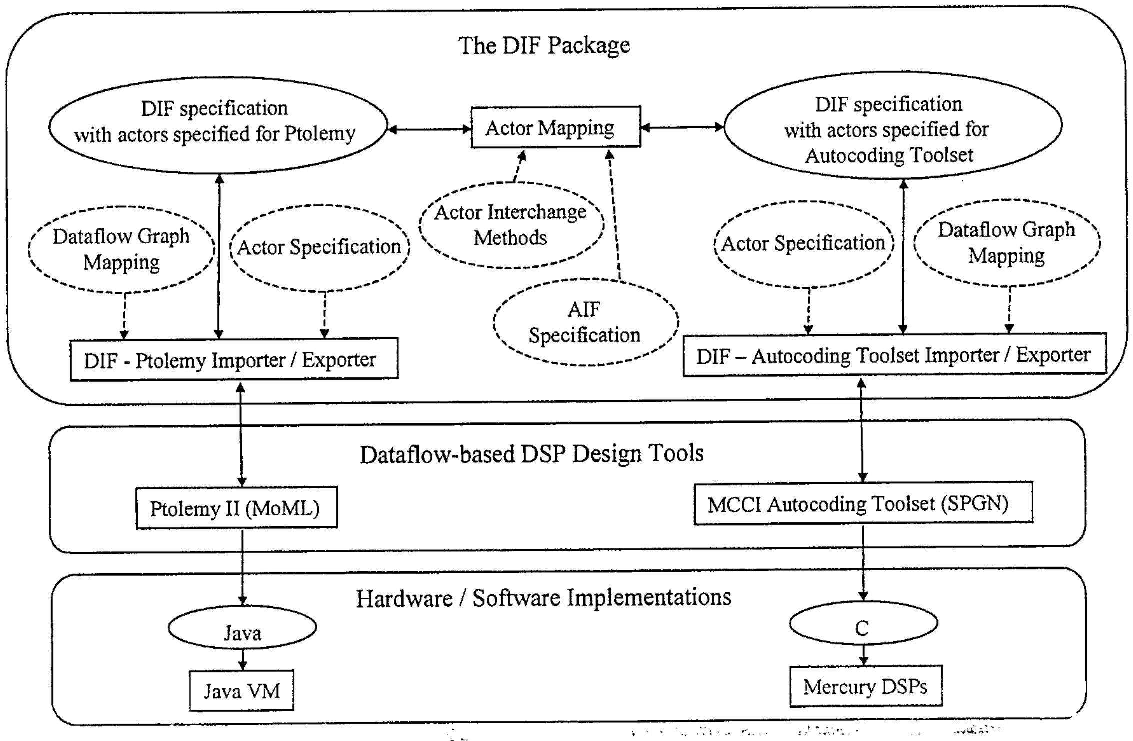

The DIF Package
DIF specification with actors specified for Ptolemy
Actor Mapping
DIF specification with actors specified for Autocoding Toolset
Dataflow Graph Mapping
Actor Specification
Actor Interchange Methods
Actor Specification
Dataflow Graph Mapping
AIF Specification
DIF - Ptolemy Importer / Exporter
DIF – Autocoding Toolset Importer / Exporter
Dataflow-based DSP Design Tools
Ptolemy II (MoML)
MCCI Autocoding Toolset (SPGN)
Hardware / Software Implementations
Java
C
Java VM
Mercury DSPs

Figure 15. The DIF Porting Mechanism.

7.1 The DIF Porting Mechanism

Figure 15 illustrates our proposed porting mechanism. It consists of three major steps: exporting, actor mapping, and importing. Let us take porting from the Autocoding Toolset to Ptolemy II as an example and introduce the porting mechanism in detail.

The first step is to export a DSP application developed in the Autocoding Toolset to the corresponding DIF specification. In this stage, the actor information (actor specifications in the DIF actor block) is specified for the Autocoding Toolset. With the DIF-Autocoding Toolset exporter/importer, this exporting process can be done automatically. The second step invokes the *actor mapping* mechanism to map DSP computational modules from Autocoding Toolset domain primitives to Ptolemy II actors. In other words, the actor mapping mechanism interchanges the tool-dependent actor information in the DIF specification. The final step is to import the DIF specification with actor information specified for Ptolemy II to the corresponding Ptolemy II graphical representation and then from the graphical representation to an equivalent Ptolemy II MoML specification. This importing process is handled by DIF-Ptolemy exporter/importer automatically.

The key advantage of using a DIF specification as an intermediate state in achieving such efficient porting of DSP applications is the comprehensive representation in the DIF language of functional semantics and component/subsystem properties that are relevant to design and implementation of DSP applications using dataflow graphs. Except for the actor block, a DIF specification for a DSP application represents the same semantic information regardless of which design tool is importing it. Such unique semantic information is an important basis for our porting mechanism, and porting DSP applications can be achieved by properly mapping the tool-dependent actor information while transferring the dataflow semantics unaltered. Actor mapping thus plays a critical role in the porting process, and the following sub-sections describe the actor mapping process in more detail.

7.2 Actor Mapping

The objective of *actor mapping* is to map an actor in a design tool to an actor or to a set of actors in another design tool while preserving the same functionality. Because different design tools usually provide different sets of actor libraries, problems may arise due to *actor absence*, *actor mismatch*, and *actor attribute mismatch*.

If a design tool does not provide the corresponding actor in its library, we encounter the *actor absence* problem. For example, Ptolemy does not provide a matrix transpose computation but the Autocoding Toolset does. If corresponding actors exist in both libraries but functionalities of those actors do not completely match, we have an instance of the *actor mismatch* problem. For example, the FFT domain primitive in the Autocoding Toolset allows designers to select the range of the output sequence, but the FFT actor in Ptolemy does not provide this function. *Actor attribute mismatch* arises when attributes are mapped between actors but the values of corresponding attributes cannot be directly interchanged. For example, the parameter *order* of the Ptolemy FFT actor specifies the FFT order, but the corresponding parameter *N* of the Autocoding Toolset FFT domain primitive specifies the length of FFT. As a result, in order to correctly map between *order* and *N*, the equation *N* = 2^*order* must be satisfied.

The actor interchange format can significantly ease the burden of actor mismatch problems by allowing a designer a convenient means for making a one-time specification of how multiple modeling components in the target design tool can construct a sub-graph such that the subgraph functionality is compatible with the source actor. In addition to providing automation in the porting process, such conversions reduce the need for users to introduce new actor definitions in the target model, thereby reducing user effort and code bloat. Similarly, actor interchange methods can solve attribute mismatch problems by evaluating a target attribute in a consistent, centrally-specified manner, based on any subset of source attribute values. For absent actors, most design tools provide ways to create actors through some sort of actor definition language. Once users determine equivalent counterparts for absent and mismatched actors, our actor mapping mechanism can take over the job cleanly and efficiently.

Figure 15 illustrates our actor mapping approach to the porting mechanism.

7.3 The Actor Interchange Format

Actor information associated with a DSP application is described in the DIF actor block by specifying a built-in *computation* attribute and other actor attributes associated with the built-in attribute types *PARAMETER*, *INPUT*, and *OUTPUT*. Specifying actor information in the DIF actor block is referred to as actor specification. In order to map actor information a from source design tool to a target design tool, the actor mapping mechanism must be able to modify actor attributes and their values in DIF specifications. How to carry out this mapping process is generally based on the provided (input) actor interchange information.

The Actor Interchange Format (AIF) is a specification format dedicated to specifying actor interchange information. The AIF syntax consists of the actor-to-actor mapping block and the actor-to-subgraph mapping block. The actor-to-actor mapping block specifies the mapping information of computations and actor attributes from a source actor (an actor in the source design tool) to a target actor (an actor in the target design tool). On the other hand, the actor-to-subgraph mapping block specifies the mapping from a source actor to a subgraph consisting of a set of actors in the target design tool and depicts the topology and interface of this subgraph. The actor-to-subgraph mapping block is designed for use when a matching standalone actor in the target tool is unavailable, inefficient or otherwise undesirable to use in the context at hand. The following subsections 7.3.1 and 7.3.2 introduce the AIF syntax and the SableCC grammar for the Actor Interchange Format is presented in Appendix B.

7.3.1 The Actor-to-Actor Mapping Block

```
actor trgActor <- srcActor | methodID(arg1, ..., argN) {
   trgAtID : type = value;
   trgAtID : type <- srcAtID : type | methodID(arg1, ..., argN);
   trgAtID1 : type, ..., trgAtIDn : type <- srcAtID : type;
   trgAtID : type <- srcAtID1 : type, ..., srcAtIDn : type;
}
```

In the first line, the keyword *actor* indicates the actor-to-actor mapping. The *srcActor* and *trgActor* specifiers designate the computations (built-in *computation* attribute) of the source actor and target actor, respectively. A method *methodID* is given optionally to specify a prior condition for this mapping (i.e., a condition that must be satisfied in order to trigger the mapping). Arguments *arg1* through *argN* can be assigned values or expressions of source actor attributes. At runtime, this method can determine whether or not the mapping should be performed based on the values of source attributes.

The AIF provides four ways to specify or map to the target attribute values, each of which corresponds to a statement in the above syntax. First, it allows users to directly assign a value *value* for a target attribute *trgAtID*. The supported value types are introduced in section 3.11. Second, a target attribute *trgAtID* can be mapped from the corresponding source attribute *srcAtID*. If *methodID* is not given in this statement, the value of *trgAtID* is directly assigned by the value of *srcAtID*. On the other hand, a method *methodID* can optionally be given to evaluate or conditionally assign the value of *trgAtID* based on the runtime values of source actor attributes. Finally, the AIF also provides syntax for one-to-multiple attribute mapping and multiple-to-one attribute mapping. For such purposes, a list of identifiers can be used as an attribute value. Note that every actor attribute can have associated with it an optionally specified *type*. For related details, see the DIF attribute blocks and DIF actor block in Section 3.8 and Section 3.10.

7.3.2 Actor to Subgraph Mapping Block

```
graph trgGraph <- srcActor | methodID(arg1, ..., argN) {
   topology {
      nodes = nodeID, ..., nodeID;
      edges = edgeID (sourceNodeID, sinkNodeID),
              ...,
              edgeID (sourceNodeID, sinkNodeID);
   }
   interface {
      inputs = portID : nodeID <- srcAtID : INPUT,
              ...,
              portID : nodeID <- srcAtID : INPUT;
      outputs = portID : nodeID <- srcAtID : OUTPUT,
              ...,
              portID : nodeID <- srcAtID : OUTPUT;
   }
   actor nodeID {
      trgAtID : type = value;
      trgAtID : type = ID;
      trgAtID : type = ID1, …, IDn;
      trgAtID : type <- srcAtID : type | methodID(arg1, ..., argN) ];
      trgAtID : type <- srcAtID1 : type, ..., srcAtIDn : type;
   }
}
```

The keyword *graph* in this context indicates the actor-to-subgraph mapping. The *trgGraph* term specifies the identifier or computation in order to invoke a component representing a subgraph in the target design tool and *srcActor* specifies the computation of the source actor. As with the actor-to-actor mapping block, a method *methodID* and its arguments can be optionally given to determine whether a triggering condition is satisfied.

The *topology* block is used to portray the topology of *trgGraph* and the *interface* block defines the interface ports of *trgGraph*. The AIF syntax for the topology and interface blocks is the same as that for the corresponding blocks in the DIF language. Moreover, the AIF allows users to specify mappings from the interface attributes, *srcAtID* with built-in type INPUT or OUTPUT, of the source actor to the interface ports of the *trgGraph*.

The actor information of every node in *trgGraph* is specified in each *actor* block. The syntax of the AIF actor block is almost the same as the DIF actor block. In addition, the AIF provides syntax to map the source actor attribute *srcAtID* to the target attribute *trgAtID* while optionally taking a method for evaluating or conditionally assigning the attribute value. Moreover, multiple-to-one attribute mapping is also supported.

7.4 Actor Interchange Methods

The methods optionally specified in the actor-to-actor mapping block and actor-to-subgraph mapping block are used to perform conditional checks or to evaluate attribute values. They are referred to as *actor interchange methods*. A set of commonly-used actor interchange methods are defined in a built-in Java class in the DIF package. Users can extend this class and design more specific interchange methods for more complicated or specialized actor mapping scenarios. Every method used in an AIF specification must be defined in this built-in class or in one of the classes derived from it. Based on the explicit classpath and the method's signature, the correct method is invoked through the Java reflection package.

There are three built-in actor interchange methods in the DIF package: 1. *ifExpression("expression")*: this method evaluates the boolean expression and returns true or false; 2. *assign("expression")*: this method evaluates the input expression and returns the evaluated value; 3. *conditionalAssign("valueExpression", "conditionalExpression")*: this method returns the value of valueExpression if the conditionalExpression is true, and throws an exception otherwise. Note that the attributes of the source actor can be used as variables in expressions and their values are used at runtime during evaluation. How to evaluate expressions is also an important issue in actor mapping. Ptolemy II provides an efficient Java package, ptolemy.data.expr, for representing variables as well as parsing and evaluating expressions; we have employed this package in the implementation of AIF.

7.5 An Actor Interchange Specification Example: FFT

Although the Autocoding Toolset and Ptolemy II both provide FFT operations, actor mismatch and attribute mismatch problems still exist between the two versions. The Autocoding Toolset FFT domain primitive has parameter *X* for data input, parameter *Y* for data output, parameter *N* for FFT length, and parameter *FI* for indicating an FFT or IFFT operation. On the other hand, the Ptolemy FFT actor has parameter *order*, input IOPort *input* and output IOPort *output*. Clearly, an actor mismatch problem arises because the FFT domain primitive provides both FFT and IFFT operations but the Ptolemy FFT actor does not. In this case, the Autocoding Toolset FFT domain primitive can be mapped to the Ptolemy FFT actor only when its parameter *FI* is not set to indicate IFFT. Moreover, an attribute mismatch problem arises because the FFT domain primitive uses the FFT length but the Ptolemy FFT actor uses the FFT order. Therefore, parameter the *N* can be mapped to the parameter *order* only when *N* = *2^order* is satisfied, where *N* and *order* are integers. The actor interchange specification for mapping the FFT operation from the Autocoding Toolset to Ptolemy II is presented in Figure 16.

The library identifier of the Autocoding Toolset FFT domain primitive is D_FFT. The classpath of the Ptolemy FFT actor is ptolemy.domains.sdf.lib.FFT. D_FFT can be mapped to ptolemy.domains.sdf.lib.FFT if the actor interchange method *ifExpression* evaluates *FI == 0* and returns true. The parameter *order* of the Ptolemy FFT actor is assigned to *log(N) / log(2)* if *log(N) / log(2)* is an integer. Therefore, the actor interchange method *conditionalAssign* evaluates and returns *log(N) / log(2)* if *(log(N)/log(2)) - rint(log(n)/log(2)) == 0* is true, where *rint()* is a round

```
actor ptolemy.domains.sdf.lib.FFT <- D_FFT | ifExpression("FI == 0") {
  order : PARAMETER <- N | conditionalAssign(
    "log(N)/log(2)","(log(N)/log(2)) - rint(log(N)/log(2)) == 0");
  input : INPUT <- X;
  output : OUTPUT <- Y;
}
```

Figure 16. The actor interchange specification of mapping the FFT operation.

to nearest integer function. Note that if *(log(N)/log(2)) - rint(log(n)/log(2)) == 0* is false, *conditionalAssign* will throw an exception indicating that the attribute mapping fails. Next, the value of parameter *X* is directly assigned to IOPort *input* for specifying the incoming edge. Similarly, the value of parameter *Y* is directly assigned to IOPort *output*.

The Autocoding Toolset FFT domain primitive also has a parameter *B*, which specifies the first point of its output sequence and a parameter *M*, which specifies the number of output points. The ability to select the range of the output sequence causes another actor mismatch problem because the Ptolemy FFT actor does not support this function. Furthermore, there is a factor of *N* difference between the Autocoding Toolset FFT domain primitive performing the IFFT operation and the Ptolemy IFFT actor. One way to solve this problem is to create a new FFT actor in Ptolemy, but it is rather time-consuming. The AIF actor-to-subgraph mapping block can be used instead to solve such actor mismatch problems by combining multiple actors in the target design tool in strategic ways to construct a subgraph such that the functionality of the subgraph is compatible to the source actor.

The actor interchange specification in Figure 17 illustrates how to map a D_FFT domain primitive with the IFFT operation and selective output length to a Ptolemy subgraph. If a D_FFT domain primitive outputs only part of its sequence, i.e., parameter *N* is not equal to parameter *M*, other Ptolemy actors are involved to extract part of the output sequence of the FFT or IFFT actors. As a result, when *FI == 1 && M != N* is true, a D_FFT domain primitive should be mapped to a Ptolemy subgraph capable of performing an IFFT operation and post-processing the output sequence. A subgraph in Ptolemy is represented by a supernode and is instantiated through the class ptolemy.actor.TypedCompositeActor.

The mapped subgraph consists of an IFFT actor, a Scale actor, a SequenceToArray actor, an ArrayExtract actor, and an ArrayToSequence actor connected in this order. The IFFT actor performs an IFFT operation, the Scale actor adjusts each sample by a factor of *N*, and the other three actors are used to extract a certain part of the output sequence. The subgraph has an input port *in* mapped from parameter *X* of D_FFT and an output port *out* mapped from parameter *Y* of D_FFT.

The classpaths of IFFT, SequenceToArray, ArrayExtract, and ArrayToSequence are specified in the *computation* attributes. Moreover, the parameter *order* of IFFT is mapped from D_FFT parameter *N* and its value is assigned *log(N) / log(2)*, if *log(N) / log(2)* is an integer. The parameter *factor* of Scale is mapped from *N*. Then, SequenceToArray converts *arrayLength* samples to an array and its parameter *arrayLength* is mapped from *N*. Next, ArrayExtract extracts *extractLength* elements starting from *sourcePosition* in the input array and puts them into an output array with length *outputArrayLength* starting from *destinationPosition*. Its parameter *sourcePosition* is mapped from D_FFT parameter *B*. Another attribute mismatch problem arises because the array starting index in Ptolemy II is 0 but it is 1 in the Autocoding Toolset. The actor interchange method *assign* solves the problem by returning $(B-1)$. Finally, ArrayToSequence converts an array to *arrayLength* samples and *arrayLength* is mapped from D_FFT parameter *M*.

```
graph ptolemy.actor.TypedCompositeActor <- D_FFT
    | ifExpression("FI == 1 && M != N") {
  topology {
   nodes = IFFT, Scale, SequenceToArray, ArrayExtract, ArrayToSequence;
   edges = e1 (IFFT, Scale), e2 (Scale, SequenceToArray),
           e3 (SequenceToArray, ArrayExtract),
           e4 (ArrayExtract, ArrayToSequence);
  }
  interface {
   inputs = in : IFFT <- X;
   outputs = out : ArrayToSequence <- Y;
  }
  actor IFFT {
   computation = "ptolemy.domains.sdf.lib.IFFT";
   order : PARAMETER <- N | conditionalAssign(
      "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
   input : INPUT = in;
   output : OUTPUT = e1;
  }
  actor Scale {
   computation = "ptolemy.actor.lib.Scale";
   input : INPUT = e1;
   output : OUTPUT = e2;
   factor : PARAMETER <- N;
  }
  actor SequenceToArray {
  computation = "ptolemy.domains.sdf.lib.SequenceToArray";
  input : INPUT = e2;
  output : OUTPUT = e3;
  arrayLength : PARAMETER <- N;
  }
  actor ArrayExtract {
   computation = "ptolemy.actor.lib.ArrayExtract";
   input : INPUT = e3;
   output : OUTPUT = e4;
   sourcePosition : PARAMETER <- B | assign("B-1");
   extractLength : PARAMETER <- M;
   destinationPosition : PARAMETER = 0;
   outputArrayLength : PARAMETER <- M;
  }
  actor ArrayToSequence {
   computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
   input : INPUT = e4;
   output : OUTPUT = out;
   arrayLength : PARAMETER <- M;
  }
 }
```

Figure 17. Actor interchange specification of actor-to-subgraph mapping of IFFT operation.

7.6 Conclusion

By supporting automatic exporting and importing for source and target design tools, the first and third porting steps are achieved. With the Actor Interchange Format and actor interchange methods for actor mapping, the entire three-step DIF porting mechanism is demonstrated, as illustrated in Figure 15.

The actor interchange methods and the corresponding AIF syntax are able to solve most attribute mismatch problems because the target attribute value can be expressed conditionally based on all source attribute values and users can design actor interchange methods for different scenarios. In addition, actor-to-subgraph mapping can solve certain actor mismatch problems because users can collect several target actors to construct a subgraph such that the functionality of the subgraph is compatible with that of the source actor. If an actor is absent, manually creating the corresponding actor is the last resort; the features in AIF greatly help to minimize the need for doing this. Once users make suitable provisions for all of the absent actors, the actor mapping mechanism associated with AIF can take over the job in an efficient, systematic fashion.

DIF is capable of porting DSP applications across dataflow-based design tools without any standard library. In this case, the Actor Interchange Format acts as a standard specification format to specify the interchange information between tools. However, cooperating with an industrial standard library for providing an actor functional interface can further facilitate the porting process. Even with a standard library, the Actor Interchange Format is still essential in mapping actors between tools and the standard library. This is a useful direction for further study in the DIF project.

8 SAR Example

In this section, we demonstrate porting a synthetic aperture radar (SAR) benchmark application from the Autocoding Toolset to Ptolemy II. This demonstration shows the effectiveness the porting mechanisms developed through DIF and AIF. The synthetic aperture radar system examined in this section was used as a benchmark in the Rapid Prototyping of Application Specific Signal Processors (RASSP) program sponsored by DARPA [16]. It represents one type of application where the processing is rather simple but the data rate is extremely high.

8.1 The SAR Application in the MCCI Autocoding Toolset

Figure 18 shows the SAR Functional Requirement developed in the MCCI Autocoding Toolset. Figure 18.(a) illustrates the top-level coarse-grain dataflow graph, *SAR_FR*. The SAR system consists of two major building blocks: range processing and azimuth processing. Passed in through the *SAR_IN* input queue, data samples are processed by node *RANGE* and node *AZIMUTH*, then they are sent to the *SAR_OUT* output queue. Node *RANGE* and node *AZIMUTH* in the *SAR_FR* graph represent the *RNG_FR* subgraph in Figure 18.(b) and the *AZI_FR* subgraph in Figure 18.(c), respectively.

Figure 18.(b) illustrates range processing in the *RNG_FR* subgraph. It consists of four nodes. Node *PAD* pads 16 zero-valued samples to the end of each 2032-sample row. Node *WEIGHT* multiplies each padded range row by a *TAYLOR_WTS* weighting sequence containing 2048 weighting values. Node *COMPRESS* performs a 2048 point Fast Fourier Transform on each range row. Node *COMPENSATE* multiplies the transformed data by the radar cross-section compensation *RCS_WTS* sequence containing 2048 compensating values.

Figure 18.(c) illustrates the azimuth processing in the *AZI_FR* subgraph. It consists of four nodes as follows. Node *CORNERTURN* corner-turns a 1024-by-2048 matrix by using matrix transpose. Node *FFT* performs a 1024-point Fast Fourier Transform on each row of the transposed matrix. Node *CONVL* multiplies each transformed row by a convolution kernel *AZ_KERN* sequence containing 1024 data values. Node *IFFT* performs an Inverse Fast Fourier Transform on the convolution result and outputs only the last 512 samples.

8.2 Porting the SAR Application to Ptolemy II

Appendix C presents the DIF specification of the Autocoding Toolset SAR application showed in Figure 18. With the actor interchange specification presented in Appendix D and the actor interchange methods developed in the DIF package, our actor mapping mechanism can translate the DIF specification in Appendix C to the DIF specification for Ptolemy actors, which is presented in Appendix E. Finally, the DIF-Ptolemy exporter/importer imports the DIF specification in Appendix E to Ptolemy II. The ported graphical representation in Ptolemy II is showed in Figure 19.

Figure 19.(a) represents the top-level coarse grain graph of the SAR application in Ptolemy II. The supernodes (blocks with red borders) RNG_FR and AZI_FR represent the range processing subgraph and the azimuth processing subgraph, respectively. Figure 19.(b) is the range processing RNG_FR graph and Figure 19.(c) is the azimuth processing graph. Node *IFFT* in Figure 18.(c) outputs only half of the IFFT sequence and there is a factor of *N* difference; actor-to-subgraph mapping is used to solve these actor mismatch problems. The node *IFFT* in Figure 18.(c) is mapped to the IFFT_SUBGRAPH in Figure 19.(d).

The MCCI Autocoding Toolset has I/O procedures specified outside of its graph specifications. As a result, we manually added I/O actors to feed data samples as well as coefficients into the SAR graph, and to write and display the results. Figure 20 shows the SAR application in Ptolemy II after adding I/O actors. The supernode SAR_FR in Figure 20 represents the top-level SAR in Figure 19.(a). Other actors in Figure 20 are used to read input samples, read coefficients, write to a file, and display the absolute value of the output waveform.

The ported SAR benchmark application in Ptolemy II works correctly. Figure 21 shows the output waveform in Ptolemy II. Figure 22 compares the output samples generated by Ptolemy II with those generated by Autocoding Toolset, and reveals that the simulation results are the same except for tolerable precision errors.

9 Conclusions and Future Work

In this report, we have introduced the DIF language version 0.2, the DIF package, and the supported dataflow models. We described our approach to automate the exporting and importing processes. Finally, we developed the DIF porting mechanism and demonstrated it through a detailed example of porting the SAR benchmark application between the MCCI Autocoding Toolset and Ptolemy II.

In ongoing and future work on the DIF project, we are extending the DIF language and the DIF package to accommodate advanced dataflow semantics such as various additional forms of dynamic graph elements and multi-mode graphs. Another useful direction for further work is integration with industrial dataflow-based design tools by combining algorithms in the DIF package with their software synthesis and code generation techniques.

10 Acknowledgements

This research was sponsored in part by DARPA (contract #F30602-01-C-0171, through the USC Information Sciences Institute and contract #DAAH01-03-C-R236 via the U.S. Army Aviation and Missile Command and through MCCI), and the Semiconductor Research Corporation (contract #2001-HJ-905).

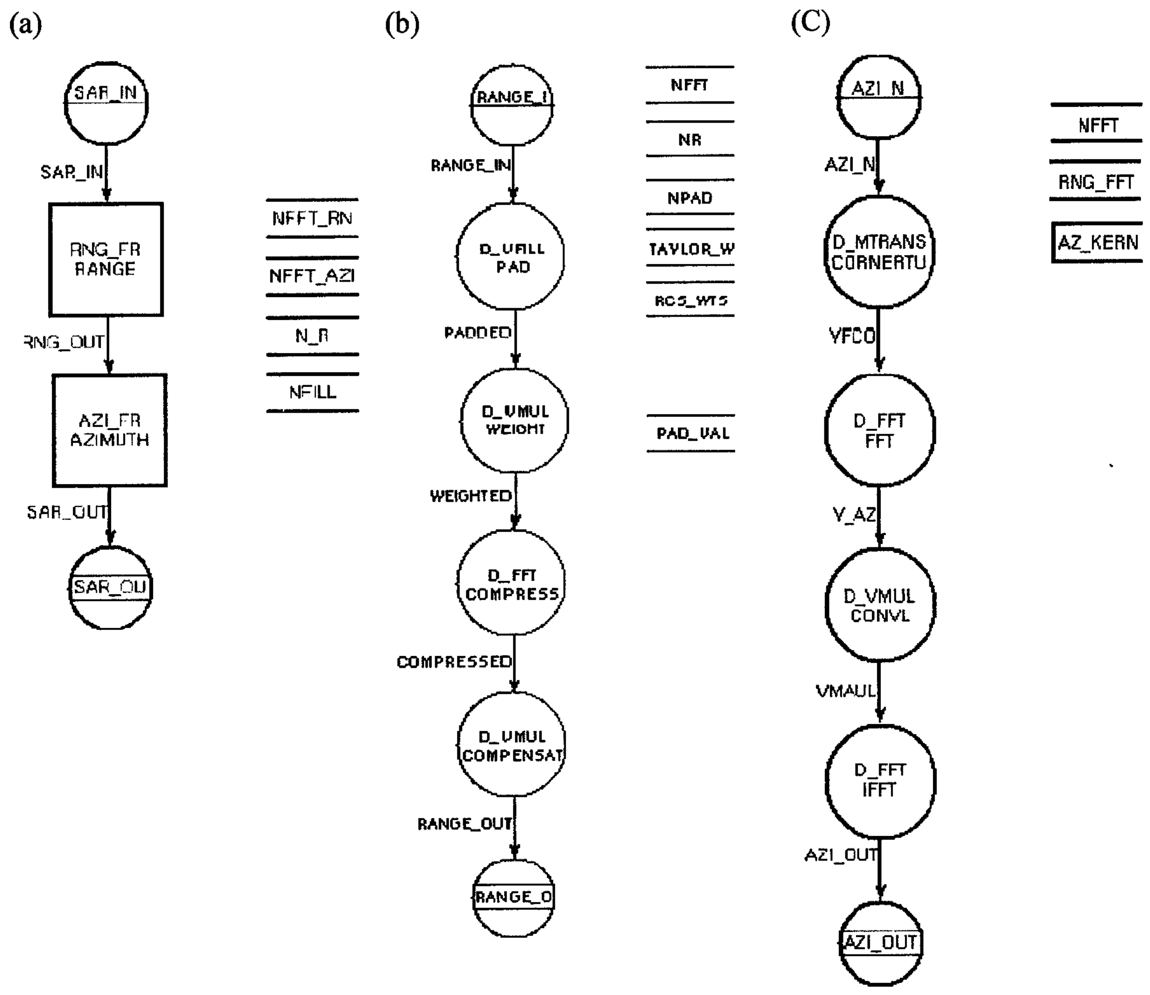


Figure 18. The SAR benchmark application in the MCCI Autocoding Toolset.
(a) SAR_FR graph. (b) RNG_FR graph. (c) AZI_FR graph.

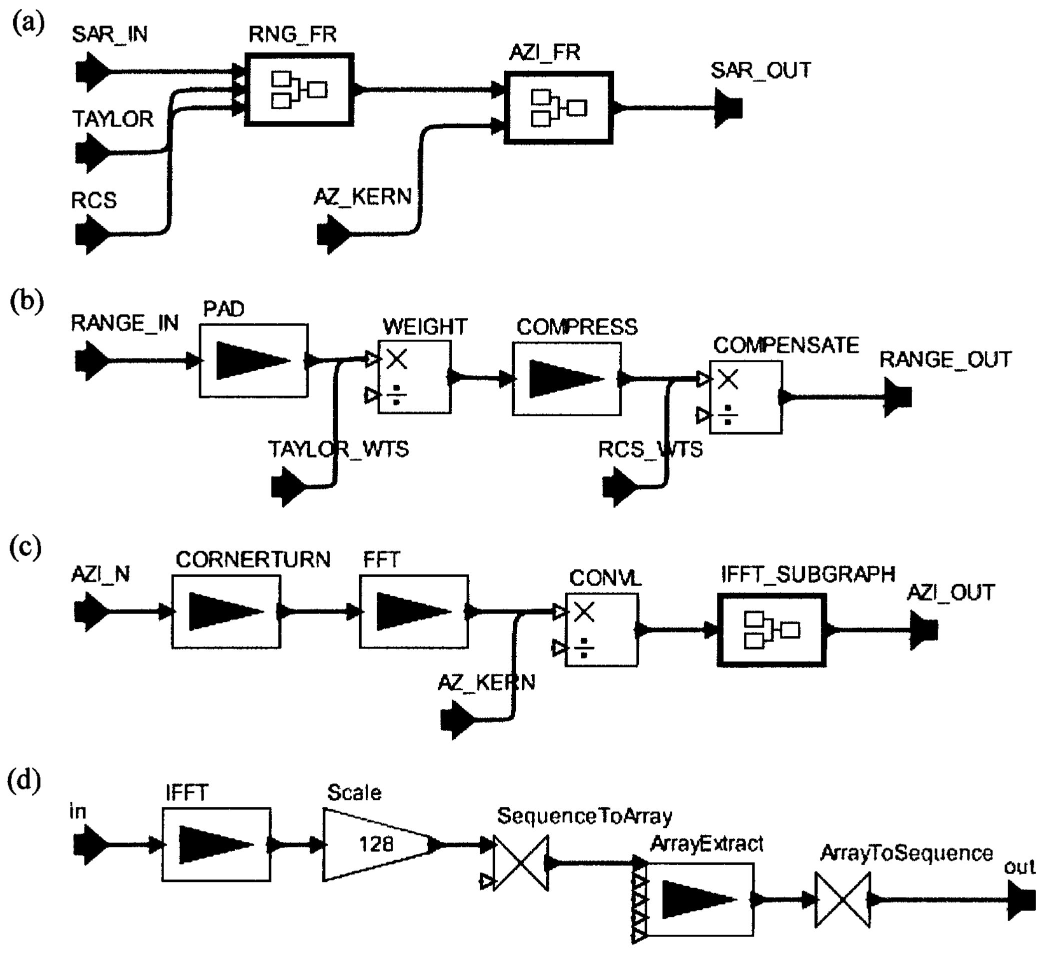


Figure 19. The SAR benchmark application in Ptolemy.
(a) SAR_FR graph. (b) RNG_FR graph. (c) AZI_FR graph. (d) IFFT_SUBGRAPH graph

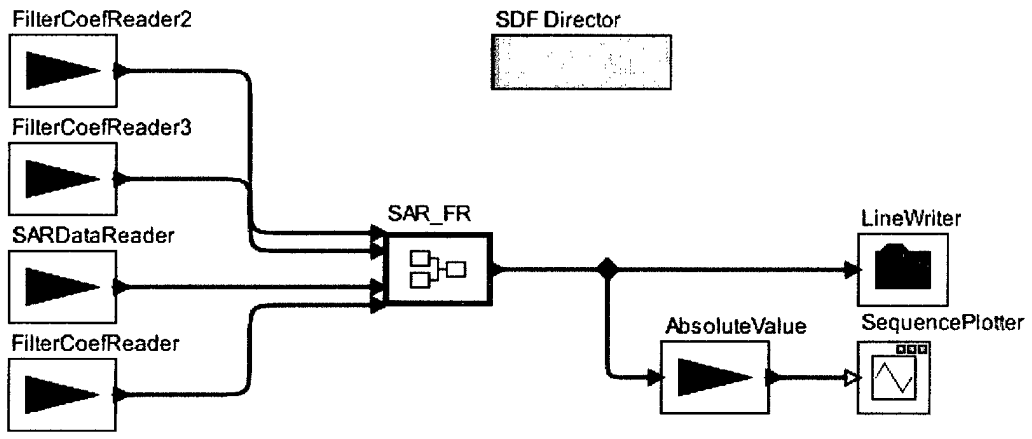


Figure 20. The SAR benchmark application in Ptolemy after adding i/o actors

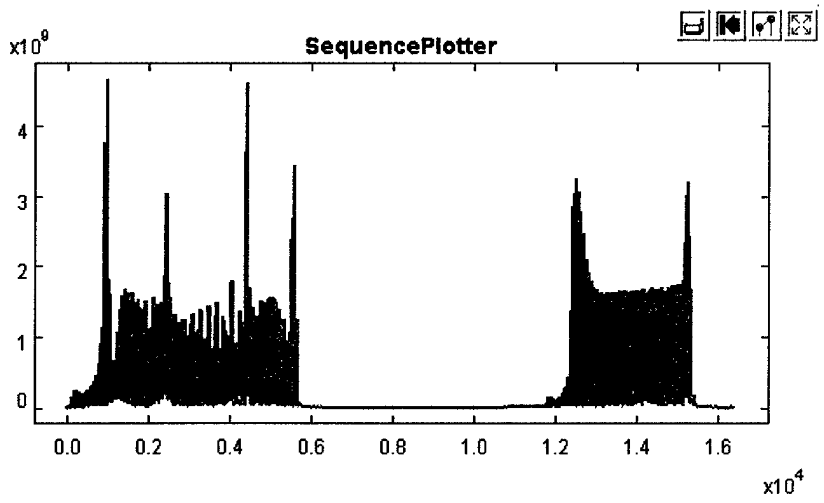
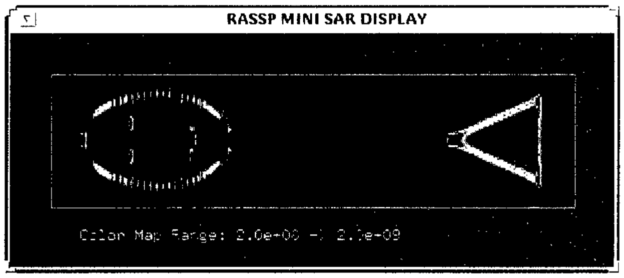


Figure 21. Simulation result of the SAR benchmark application in Ptolemy II.

Ptolemy(PGM Export)

```
1.113328370318E9, -5.672582199684E8
1.686243152456E9, -1.132239286739E9
2.280892492213E9, -1.837179778052E9
2.787030647091E9, -2.565079199379E9
3.121469726315E9, -3.124321013999E9
3.235633491442E9, -3.339997173742E9
3.126105298721E9, -3.132702116709E9
2.795907223687E9, -2.578937710771E9
2.292518065694E9, -1.852489499236E9
1.698661416987E9, -1.145532955647E9
```



MCCI(DIF Import)

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```

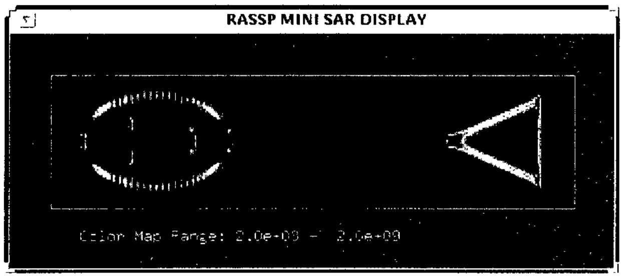


Figure 22. Simulation results of the SAR benchmark application in Ptolemy II and MCCI Autocoding Toolset.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408-2421, October 2001.

[2] S. S. Bhattacharyya. Hardware/software co-synthesis of DSP systems. In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333-378. Marcel Dekker, Inc., 2002.

[3] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151-166, June 1999.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Proc. ICASSP*, pages 3255-3258, May 1995.

[6] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Tech. Report UCB/ERL 93/69, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[7] E. Gagnon. *SableCC, an object-oriented compiler framework*. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, March 1998.

[8] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. DIF: An interchange format for dataflow-based design tools. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004. To appear.

[9] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*. Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.

[10] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng, (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture)*. Technical Memorandum UCB/ERL M03/28, University of California, Berkeley, CA USA 94720, July 16, 2003.

[11] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*. Technical Memorandum UCB/ERL M03/29, University of California, Berkeley, CA USA 94720, July 16, 2003.

[12] F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. First version of a dataflow interchange format. Technical Report UMIACS-TR-2002-98, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2002. Also Computer Science Technical Report CS-TR-4418.

[13] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987..

[14] Keshab K. Parhi. *VLSI Digital Signal Processing Systems, Design and Implementation*, John Wiley, 1999.

[15] C. B. Robbins. *Using The MCCI Autocoding Toolset Overview*, Document Version 0.98a, Management, Communications & Control, Inc.

[16] C. B. Robbins. *Using The MCCI Autocoding Toolset Tutorial*. Document Version 0.9a, Management, Communications & Control, Inc.

[17] J. Teich and S. S. Bhattacharyya. Analysis of dataflow programs with interval-limited data-rates. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 507-518, Samos, Greece, July 2004.

Appendix A

The SableCC (version 2.16.2) grammar of the Dataflow Interchange Format.

```
Package mapss.dif.language.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';

  graph = 'graph';
  attribute = 'attribute';
  basedon = 'basedon';
  interface = 'interface';
  parameter = 'parameter';
  refinement = 'refinement';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
   dot_identifier  =  non_digit  (digit  |  non_digit)*  ('.'  non_digit  (digit  |
non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  graph_list = graph_block*;
  graph_block = identifier name l_bkt block* r_bkt;
  block =
    {basedon}                    basedon basedon_body |
    {topology}                   topology topology_body |
    {interface}                  interface interface_body |
    {parameter}                  parameter parameter_body |
    {refinement}                 refinement refinement_body |
    {builtin_attribute}          identifier attribute_body |
    {user_defined_attribute}     attribute name attribute_body |
    {actor}                      actor name actor_body;

  name = {identifier} identifier | {string_identifier} string_identifier;

  /*************************************
   * Definitions for basedon block:
```

```
 */

basedon_body = l_bkt basedon_expression r_bkt;
basedon_expression = name semicolon;

/*************************************
 * Definitions for topology block:
 */

topology_body = l_bkt topology_list* r_bkt;
topology_list =
  {nodes}  nodes equal name node_identifier_tail* semicolon |
  {edges}  edges equal edge_definition edge_definition_tail* semicolon ;

node_identifier_tail = comma name;
edge_definition = [edge]:name l_par
                  [source]:name comma
                  [sink]:name r_par;
edge_definition_tail = comma edge_definition;

/*************************************
 * Definitions for interface block:
 */

interface_body = l_bkt interface_expression* r_bkt;
interface_expression =
  {input}   inputs equal port_definition port_definition_tail* semicolon |
  {output}  outputs equal port_definition port_definition_tail* semicolon;

port_definition = {plain} name |
                  {node}  [port]:name colon [node]:name;
port_definition_tail = comma port_definition;

/*************************************
 * Definitions for parameter block:
 */

parameter_body = l_bkt parameter_expression* r_bkt;
parameter_expression =
  {value}     name equal value semicolon |
  {range}     name colon range_block semicolon |
  {blank}     name semicolon;

range_block = range range_tail*;
range =
  {closed_closed}     l_sqr [left]:number comma [right]:number r_sqr |
  {open_closed}       l_par [left]:number comma [right]:number r_sqr |
  {closed_open}       l_sqr [left]:number comma [right]:number r_par |
  {open_open}         l_par [left]:number comma [right]:number r_par |
  {discrete}          l_bkt number discrete_range_number_tail* r_bkt;
discrete_range_number_tail = comma number;
range_tail = plus range;
number = {double} double | {integer} integer;
```

```
/************************************
 * Definitions for refinement block:
 */

refinement_body = l_bkt refinement_definition refinement_expression* r_bkt;
refinement_definition = [graph]:name equal [node]:name semicolon;
refinement_expression =
  {ports}     [port]:name colon [element]:name semicolon |
  {params}    [subparam]:name equal [param]:name semicolon;

/************************************
 * Definitions for attribute block:
 */

attribute_body = l_bkt attribute_expression* r_bkt;
attribute_expression =
  {value} name? equal value semicolon |
  {reference} [element]:name? equal [reference]:name semicolon |
  {subelement_assign} [trggraph]:name [fst]:dot [trgele]:name equal
                      [srcgraph]:name [snd]:dot [srcele]:name semicolon |
  {idlist} name? equal id_list semicolon;

id_list = name ref_id_tail+;
ref_id_tail = comma name;

/************************************
 * Definitions for actor block:
 */

actor_body = l_bkt actor_expression* r_bkt;
actor_expression =
  {value} name type? equal value semicolon |
  {reference} [argument]:name type? equal [reference]:name semicolon |
  {reflist} name type? equal id_list semicolon;

type =
  {identifier} colon identifier |
  {dot_identifier} colon dot_identifier;

/************************************
 * Definitions for value:
 */

value =
  {integer} integer |
  {double} double |
  {complex} l_par [real]:double comma [imag]:double r_par |
  {int_matrix} l_sqr int_row int_row_tail* r_sqr |
  {double_matrix} l_sqr double_row double_row_tail* r_sqr |
  {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
  {string} concatenated_string_value |
  {boolean} boolean_value |
  {array} l_bkt value value_tail* r_bkt;
```

```
int_row = integer integer_tail*;
integer_tail = comma integer;
int_row_tail = semicolon int_row;

double_row = double double_tail*;
double_tail = comma double;
double_row_tail = semicolon double_row;

complex = l_par [real]:double comma [imag]:double r_par;
complex_row = complex complex_tail*;
complex_tail = comma complex;
complex_row_tail = semicolon complex_row;

concatenated_string_value = string string_tail*;

boolean_value =
  {true} true |
  {false} false;

value_tail = comma value;
```

Appendix B

The SableCC (version 2.16.2) grammar of the Actor Interchange Format.

```
Package mapss.dif.aif.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';
  map_to = '->';
  map_from = '<-';
  given_that = '|';

  graph = 'graph';
  interface = 'interface';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
  dot_identifier = non_digit (digit | non_digit)*
                   ('.' non_digit (digit | non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  aif_list = aif_block*;
  aif_block = {actor} actor [trg]:type map_from [src]:type
                      method_expression? l_bkt attribute_body* r_bkt |
              {graph} graph [trg]:type map_from [src]:type
                      method_expression? l_bkt block* r_bkt;

  type = {identifier} identifier |
         {dot_identifier} dot_identifier;
  method_expression = given_that identifier
                      l_par argument argument_tail* r_par;
  argument = {id} identifier |
             {value} value;
  argument_tail = comma argument;

  /*************************************
   * Definitions for attribute body:
   */
```

```
attribute_body = {mapping} attribute_mapping |
                 {assign} attribute_assign;
attribute_assign = attribute equal value semicolon;
attribute_mapping = {single} [trg]:attribute map_from [src]:attribute
                              method_expression? semicolon |
                    {multi_to_one} attribute map_from attributes semicolon |
                    {one_to_multi} attributes map_from attribute semicolon;

attributes = attribute attribute_tail+;
type_expression = colon type;
attributes = attribute attribute_tail+;
attribute_tail = comma attribute;

/**************************************
 * Definitions for block:
 */

block = {topology} topology topology_body |
        {interface} interface interface_body |
        {actor} actor name actor_body;

name = {identifier} identifier |
       {string_identifier} string_identifier;

/**************************************
 * Definitions for topology block:
 */

topology_body = l_bkt topology_list* r_bkt;
topology_list =
  {nodes}  nodes equal name node_identifier_tail* semicolon |
  {edges}  edges equal edge_definition edge_definition_tail* semicolon ;

node_identifier_tail = comma name;
edge_definition = [edge]:name l_par
                  [source]:name comma
                  [sink]:name r_par;
edge_definition_tail = comma edge_definition;

/**************************************
 * Definitions for interface block:
 */

interface_body = l_bkt interface_expression* r_bkt;
interface_expression =
  {input}   inputs equal port_definition port_definition_tail* semicolon |
  {output}  outputs equal port_definition port_definition_tail* semicolon;

port_definition = {plain} name port_mapping? |
                  {node}  [port]:name colon [node]:name port_mapping?;
port_definition_tail = comma port_definition;
port_mapping = map_from attribute;
```

```
/************************************
 * Definitions for actor block:
 */

actor_body = l_bkt actor_expression* r_bkt;
actor_expression =
  {value} name type_expression? equal value semicolon |
  {reference} [argument]:name type_expression? equal
              [reference]:name semicolon |
  {map} name type_expression? map_from attribute
        method_expression? semicolon |
  {multi_map} name type_expression? map_from attributes semicolon |
  {reflist} name type_expression? equal id_list semicolon;

id_list = name ref_id_tail+;
ref_id_tail = comma name;

/************************************
 * Definitions for value:
 */

value =
  {integer} integer |
  {double} double |
  {complex} l_par [real]:double comma [imag]:double r_par |
  {int_matrix} l_sqr int_row int_row_tail* r_sqr |
  {double_matrix} l_sqr double_row double_row_tail* r_sqr |
  {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
  {string} concatenated_string_value |
  {boolean} boolean_value |
  {array} l_bkt value value_tail* r_bkt;

int_row = integer integer_tail*;
integer_tail = comma integer;
int_row_tail = semicolon int_row;

double_row = double double_tail*;
double_tail = comma double;
double_row_tail = semicolon double_row;

complex = l_par [real]:double comma [imag]:double r_par;
complex_row = complex complex_tail*;
complex_tail = comma complex;
complex_row_tail = semicolon complex_row;

concatenated_string_value = string string_tail*;

boolean_value =
  {true} true |
  {false} false;

value_tail = comma value;
```

Appendix C

The DIF specification of the MCCI SAR benchmark application.

```
dif RNG_FR {
  topology {
    nodes = PAD, WEIGHT, COMPRESS, COMPENSATE;
    edges = PADDED (PAD, WEIGHT),
            WEIGHTED (WEIGHT, COMPRESS),
            COMPRESSED (COMPRESS, COMPENSATE);
  }
  interface {
    inputs = RANGE_IN : PAD, TAYLOR_WTS : WEIGHT, RCS_WTS : COMPENSATE;
    outputs = RANGE_OUT : COMPENSATE;
  }
  parameter {
    NFFT;
    NR;
    NPAD;
    PAD_VAL = (0.0, 0.0);
  }
  actor PAD {
    computation = "D_VFILL" ;
    N = NR;
    P = NPAD;
    V = PAD_VAL;
    X = RANGE_IN;
    Y = PADDED;
  }
  actor WEIGHT {
    computation = "D_VMUL";
    N = NFFT;
    X = PADDED;
    Y = TAYLOR_WTS;
    Z = WEIGHTED;
  }
  actor COMPRESS {
    computation = "D_FFT";
    N = NFFT;
    FI = 0;
    X = WEIGHTED;
    Y = COMPRESSED;
  }
  actor COMPENSATE {
    computation = "D_VMUL";
    N = NFFT;
    X = COMPRESSED;
    Y = RCS_WTS;
    Z = RANGE_OUT;
  }
}

dif AZI_FR {
    topology {
      nodes = CORNERTURN, FFT, CONVL, IFFT;
      edges = YFCO (CORNERTURN, FFT),
              Y_AZ (FFT, CONVL),
              VMAUL (CONVL, IFFT);
    }
    interface {
      inputs = AZI_N : CORNERTURN, AZ_KERN : CONVL;
      outputs = AZI_OUT : IFFT;
    }
    parameter {
      NFFT;
      RNG_FFT;
    }
    actor CORNERTURN {
      computation = "D_MTRAN";
      M = NFFT;
      N = RNG_FFT;
      X = AZI_N;
      Y = YFCO;
    }
    actor FFT {
      computation = "D_FFT";
      N = NFFT;
      FI = 0;
      X = YFCO;
      Y = Y_AZ;
    }
    actor CONVL {
      computation = "D_VMUL";
      N = NFFT;
      X = Y_AZ;
      Y = AZ_KERN;
      Z = VMAUL;
    }
    actor IFFT {
      computation = "D_FFT";
      N = NFFT;
      FI = 1;
      X = VMAUL;
      Y = AZI_OUT;
      M = "NFFT/2";
      B = "(NFFT/2)+1";
    }
  }

  dif FR_SAR {
    topology {
      nodes = RANGE, AZIMUTH;
      edges = RNG_OUT (RANGE, AZIMUTH);
    }
    interface {
      inputs = SAR_IN : RANGE, TAYLOR : RANGE, RCS : RANGE, AZ_KERN : AZIMUTH;
      outputs = SAR_OUT : AZIMUTH;
    }
    parameter {
      NFFT_RNG = 256;
      NFFT_AZI = 128;
      N_R = 235;
      NFILL = "NFFT_RNG-N_R";
    }
    refinement {
      RNG_FR = RANGE;
      RANGE_IN : SAR_IN;
      RANGE_OUT : RNG_OUT;
      TAYLOR_WTS : TAYLOR;
      RCS_WTS : RCS;
      NFFT = NFFT_RNG;
      NR = N_R;
      NPAD = NFILL;
    }
    refinement {
      AZI_FR = AZIMUTH;
      AZI_N : RNG_OUT;
      AZI_OUT : SAR_OUT;
      AZ_KERN : AZ_KERN;
      NFFT = NFFT_AZI;
      RNG_FFT = NFFT_RNG;
    }
    actor RANGE {
      computation = "SUBGRAPH";
    }
    actor AZIMUTH {
      computation = "SUBGRAPH";
    }
  }
```

Appendix D

The actor interchange specification for Autocoding Toolset to Ptolemy II actor mapping.

```
graph ptolemy.actor.TypedCompositeActor <- D_FFT
        | ifExpression("FI == 1 && M != N") {
    topology {
        nodes = IFFT, Scale, SequenceToArray, ArrayExtract, ArrayToSequence;
        edges = e1 (IFFT, Scale),
                e2 (Scale, SequenceToArray),
                e3 (SequenceToArray, ArrayExtract),
                e4 (ArrayExtract, ArrayToSequence);
    }
    interface {
        inputs = in : IFFT <- X;
        outputs = out : ArrayToSequence <- Y;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order : PARAMETER <- N | conditionalAssign(
            "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
        input : INPUT = in;
        output : OUTPUT = e1;
    }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        input : INPUT = e1;
        output : OUTPUT = e2;
        factor : PARAMETER <- N;
    }
    actor SequenceToArray {
        computation = "ptolemy.domains.sdf.lib.SequenceToArray";
        input : INPUT = e2;
        output : OUTPUT = e3;
        arrayLength : PARAMETER <- N;
    }
    actor ArrayExtract {
        computation = "ptolemy.actor.lib.ArrayExtract";
        input : INPUT = e3;
        output : OUTPUT = e4;
        sourcePosition : PARAMETER <- B | assign("B-1");
        extractLength : PARAMETER <- M;
        destinationPosition : PARAMETER = 0;
        outputArrayLength : PARAMETER <- M;
    }
    actor ArrayToSequence {
        computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
        input : INPUT = e4;
        output : OUTPUT = out;
        arrayLength : PARAMETER <- M;
    }
}
```

```
actor ptolemy.domains.sdf.lib.FFT <- D_FFT | ifExpression("FI == 0") {
    order : PARAMETER <- N | conditionalAssign(
        "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.domains.sdf.lib.IFFT <- D_FFT | ifExpression("FI == 1") {
    order : PARAMETER <- N | conditionalAssign(
        "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT <- X;
    output : OUTPUT <- Y;
}


actor mapss.applications.sar.MatrixTranspose <- D_MTRAN {
    rowN : PARAMETER <- M;
    colN : PARAMETER <- N;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.lib.MultiplyDivide <- D_VMUL {
    multiply : INPUT <- X, Y;
    output : OUTPUT <- Z;
}

actor mapss.applications.sar.SequencePad <- D_VFILL {
    inputLength : PARAMETER <- N;
    outputLength : PARAMETER <- P | assign("P+N");
    padValue : PARAMETER <- V;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.TypedCompositeActor <- SUBGRAPH {}
```

Appendix E

The DIF specification of the ported SAR benchmark application in Ptolemy II.

```
dif IFFT_SUBGRAPH {
    topology {
        nodes = IFFT,
                Scale,
                SequenceToArray,
                ArrayExtract,
                ArrayToSequence;
        edges = e1 (IFFT, Scale),
                e2 (Scale, SequenceToArray),
                e3 (SequenceToArray, ArrayExtract),
                e4 (ArrayExtract, ArrayToSequence);
    }
    interface {
        inputs = in:IFFT;
        outputs = out:ArrayToSequence;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order : PARAMETER = 7.0;
        input : INPUT = in;
        output : OUTPUT = e1;
    }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        input : INPUT = e1;
        output : OUTPUT = e2;
        factor : PARAMETER = 128;
    }
    actor SequenceToArray {
        computation = "ptolemy.domains.sdf.lib.SequenceToArray";
        input : INPUT = e2;
        output : OUTPUT = e3;
        arrayLength : PARAMETER = 128;
    }
    actor ArrayExtract {
        computation = "ptolemy.actor.lib.ArrayExtract";
        input : INPUT = e3;
        output : OUTPUT = e4;
        sourcePosition : PARAMETER = 64;
        extractLength : PARAMETER = 64;
        destinationPosition : PARAMETER = 0;
        outputArrayLength : PARAMETER = 64;
    }
    actor ArrayToSequence {
        computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
        input : INPUT = e4;
        output : OUTPUT = out;
        arrayLength : PARAMETER = 64;
    }
```

```
}

dif RNG_FR {
    topology {
        nodes = PAD,
                WEIGHT,
                COMPRESS,
                COMPENSATE;
        edges = PADDED (PAD, WEIGHT),
                WEIGHTED (WEIGHT, COMPRESS),
                COMPRESSED (COMPRESS, COMPENSATE);
    }
    interface {
        inputs = RANGE_IN:PAD,
                 TAYLOR_WTS:WEIGHT,
                 RCS_WTS:COMPENSATE;
        outputs = RANGE_OUT:COMPENSATE;
    }
    parameter {
        NFFT;
        NR;
        NPAD;
        PAD_VAL = (0.0,0.0);
    }
    actor PAD {
        computation = "mapss.applications.sar.SequencePad";
        inputLength : PARAMETER = NR;
        outputLength : PARAMETER = 256;
        padValue : PARAMETER = PAD_VAL;
        input : INPUT = RANGE_IN;
        output : OUTPUT = PADDED;
    }
    actor WEIGHT {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = PADDED, TAYLOR_WTS;
        output : OUTPUT = WEIGHTED;
    }
    actor COMPRESS {
        computation = "ptolemy.domains.sdf.lib.FFT";
        order : PARAMETER = 8.0;
        input : INPUT = WEIGHTED;
        output : OUTPUT = COMPRESSED;
    }
    actor COMPENSATE {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = COMPRESSED, RCS_WTS;
        output : OUTPUT = RANGE_OUT;
    }
}

dif AZI_FR {
    topology {
        nodes = CORNERTURN,
                FFT,
                  CONVL,
                  IFFT;
         edges = YFCO (CORNERTURN, FFT),
                  Y_AZ (FFT, CONVL),
                  VMAUL (CONVL, IFFT);
     }
     interface {
         inputs = AZI_N:CORNERTURN,
                   AZ_KERN:CONVL;
         outputs = AZI_OUT:IFFT;
     }
     parameter {
         NFFT;
         RNG_FFT;
     }
     refinement {
         IFFT_SUBGRAPH = IFFT;
         in : VMAUL;
         out : AZI_OUT;
     }
     actor CORNERTURN {
         computation = "mapss.applications.sar.MatrixTranspose";
         rowN : PARAMETER = NFFT;
         colN : PARAMETER = RNG_FFT;
         input : INPUT = AZI_N;
         output : OUTPUT = YFCO;
     }
     actor FFT {
         computation = "ptolemy.domains.sdf.lib.FFT";
         order : PARAMETER = 7.0;
         input : INPUT = YFCO;
         output : OUTPUT = Y_AZ;
     }
     actor CONVL {
         computation = "ptolemy.actor.lib.MultiplyDivide";
         multiply : INPUT = Y_AZ, AZ_KERN;
         output : OUTPUT = VMAUL;
     }
     actor IFFT {
         computation = "ptolemy.actor.TypedCompositeActor";
     }
}

dif FR_SAR {
     topology {
         nodes = RANGE,
                  AZIMUTH;
         edges = RNG_OUT (RANGE, AZIMUTH);
     }
     interface {
         inputs = SAR_IN:RANGE,
                   TAYLOR:RANGE,
                   RCS:RANGE,
                   AZ_KERN:AZIMUTH;
```

```
            outputs = SAR_OUT:AZIMUTH;
        }
        parameter {
            NFFT_RNG = 256;
            NFFT_AZI = 128;
            N_R = 235;
            NFILL = "NFFT_RNG-N_R";
        }
        refinement {
            RNG_FR = RANGE;
            RANGE_IN : SAR_IN;
            TAYLOR_WTS : TAYLOR;
            RCS_WTS : RCS;
            RANGE_OUT : RNG_OUT;
            NFFT = NFFT_RNG;
            NR = N_R;
            NPAD = NFILL;
        }
        refinement {
            AZI_FR = AZIMUTH;
            AZI_N : RNG_OUT;
            AZ_KERN : AZ_KERN;
            AZI_OUT : SAR_OUT;
            NFFT = NFFT_AZI;
            RNG_FFT = NFFT_RNG;
        }
        actor RANGE {
            computation = "ptolemy.actor.TypedCompositeActor";
        }
        actor AZIMUTH {
            computation = "ptolemy.actor.TypedCompositeActor";
        }
    }
```

Appendix B to Final Report on

DIF - A Language for Dataflow Graph Specification and Exchange

October 27, 2004

Sponsored by

Defense Advanced Research Projects Agency (DOD)
(Controlling DARPA Office)

ARPA Order C043/70

Issued by U.S. Army Aviation and Missile Command Under

DAAH01-03-C-R236

Table of Contents
Graph Listings

Top Level SAR Graph SPGN

```
%% Functional Requirements Graph for RASSP SAR example
%%
%% One Polarization, full processing
%%
%% Carl Ecklund
%% MCCI
%%
%GRAPH( FR_SAR
  GIP    =
        %% Taylor weighting to reduce sidelobes of
        %% compressed pulse.
        %%
        TAYLOR : CFLOAT ARRAY(2048),

        %% Compressed pulse radar cross section weights.
        %%
        RCS : FLOAT ARRAY(2048)
  VAR    = %% Azimuth convolution kernel - Selection based on slant range.
        %% Total of 31 kernels.  16 used in processing one frame.
        %% Selection is based on range row being processed.  Same
        %% kernel is used for 128 rows.
        AZ_KERN : CFLOAT ARRAY(1024)
  INPUTQ  =
        %% Complex integer data that has been FIR filtered
        %%
        %%   SAR_IN : CINT

        %% using cfloat data set
        SAR_IN : CFLOAT
  OUTPUTQ =
        %% Processed data out
        %%
        SAR_OUT : CFLOAT )
%GIP( NFFT_RNG : INT INITIALIZE TO 2048 )
%GIP( NFFT_AZI : INT INITIALIZE TO 1024 )
%GIP( N_R : INT INITIALIZE TO 2032 )
%GIP( NFILL : INT INITIALIZE TO NFFT_RNG-N_R )
%QUEUE( RNG_OUT : CFLOAT INITIALIZE TO (NFFT_RNG*NFFT_AZI)/2 OF
  <0.0E0,0.0E0> )
%% Graph RNG_FR implements functional requirements for range processing
%%
%SUBGRAPH( RANGE
  GRAPH   = RNG_FR
  GIP     = NFFT_RNG,
        N_R,
        NFILL,
        TAYLOR,
        RCS
  INPUTQ  = SAR_IN
  OUTPUTQ = RNG_OUT )

%% Graph AZI_FR implements functional requirements for azimuth processing
%%
%SUBGRAPH( AZIMUTH
  GRAPH   = AZI_FR
  GIP     = NFFT_AZI,
          NFFT_RNG
  VAR     = AZ_KERN
  INPUTQ  = RNG_OUT
  OUTPUTQ = SAR_OUT )
%ENDGRAPH
```

SAR RANGE Processing Graph SPGN

```
%% Functional requirements graph for range processing of
%% SAR example.
%%
%GRAPH( RNG_FR
  GIP    = NFFT : INT,
         NR : INT,
         NPAD : INT,
         TAYLOR_WTS : CFLOAT ARRAY(2048),
         RCS_WTS : FLOAT ARRAY(2048)
  INPUTQ  = %%  use cfloat data for initial verification
         %%  RANGE_IN : CINT
         RANGE_IN : CFLOAT
  OUTPUTQ = RANGE_OUT : CFLOAT )
%GIP( PAD_VAL : CFLOAT INITIALIZE TO <0.0E0,0.0E0> )
%GIP( NRNG : INT INITIALIZE TO 512 )
%QUEUE( PADDED : CFLOAT )
%QUEUE( WEIGHTED : CFLOAT )
%QUEUE( COMPRESSED : CFLOAT )

%% Pad range row to NFFT_RNG size
%%
%NODE( PAD
  PRIMITIVE = D_VFILL
  PRIM_IN   = NR,
          NPAD,
          UNUSED,
          PAD_VAL,
            %% for initial testing use cfloat data into graph
            %%  conversion node not required!
            %% pipe cfloat data in directly to fill
            %%   CONVERTED THRESHOLD = NR*NRNG
          RANGE_IN THRESHOLD = NRNG*NR
  PRIM_OUT  = PADDED )

%% Weight using Taylor function prior to FFT.
%%
%NODE( WEIGHT
  PRIMITIVE = D_VMUL
  PRIM_IN   = NFFT,
          UNUSED,
          PADDED THRESHOLD = NRNG*NFFT,
          TAYLOR_WTS
  PRIM_OUT  = WEIGHTED )

%NODE( COMPRESS
  PRIMITIVE = D_FFT
  PRIM_IN   = NFFT,
          UNUSED,
          UNUSED,
          UNUSED,
          UNUSED,
          WEIGHTED THRESHOLD = NRNG*NFFT
  PRIM_OUT  = COMPRESSED )
```

```
%% RCS compensation
%%
%NODE( COMPENSATE
  PRIMITIVE = D_VMUL
  PRIM_IN   = NFFT,
            UNUSED,
            COMPRESSED THRESHOLD = NRNG*NFFT,
            RCS_WTS
  PRIM_OUT  = RANGE_OUT )
%ENDGRAPH
```

SAR AZIMUTH Processing Graph SPGN

```
%% Functional Requirements for azimuth processing
%%
%GRAPH( AZI_FR
  GIP     = NFFT : INT,
          RNG_FFT : INT
  VAR     = AZ_KERN : CFLOAT ARRAY(1024)
  INPUTQ  = AZI_N : CFLOAT
  OUTPUTQ = AZI_OUT : CFLOAT )

%QUEUE( YFCO : CFLOAT )
%QUEUE( Y_AZ : CFLOAT )
%QUEUE( VMAUL : CFLOAT )

%% Cornerturn the data using a matrix transpose operation.
%%
%NODE( CORNERTURN
  PRIMITIVE = D_MTRANS
  PRIM_IN   = NFFT,
            RNG_FFT,
            AZI_N THRESHOLD = NFFT*RNG_FFT
                CONSUME = NFFT*RNG_FFT/2
  PRIM_OUT  = YFCO )
%NODE( FFT
  PRIMITIVE = D_FFT
  PRIM_IN   = NFFT,
            UNUSED,
            UNUSED,
            UNUSED,
            UNUSED,
            YFCO THRESHOLD = NFFT*RNG_FFT
  PRIM_OUT  = Y_AZ )
%NODE( CONVL
  PRIMITIVE = D_VMUL
  PRIM_IN   = NFFT,
            UNUSED,
            Y_AZ THRESHOLD = NFFT*RNG_FFT,
            AZ_KERN
  PRIM_OUT  = VMAUL )
%NODE( IFFT
  PRIMITIVE = D_FFT
  PRIM_IN   = NFFT,
            NFFT/2,
            1,
            (NFFT/2)+1,
            UNUSED,
            VMAUL THRESHOLD = NFFT*RNG_FFT
  PRIM_OUT  = AZI_OUT )
%ENDGRAPH
```

DIF SAR Graph

```
dif FR_SAR {
  topology {
    nodes = RANGE, AZIMUTH;
    edges = RNGOUT (RANGE, AZIMUTH);
  }
  interface {
    inputs = SAR_IN : RANGE,
             TAYLOR : RANGE,
             RCS : RANGE,
             AZKERN : AZIMUTH;
    outputs = SAR_OUT : AZIMUTH;
  }
  parameter {
    NFFT_RNG = 2048;
    NFFT_AZI = 1024;
    N_R = 2032;
    NFILL = 16;
  }
  refinement {
    RNG_FR = RANGE;
    RANGE_IN : SAR_IN;
    RANGE_OUT : RNGOUT;
    TAYLOR_WTS : TAYLOR;
    RCS_WTS : RCS;
    NFFT =  NFFT_RNG;
    NR = N_R;
    NPAD = NFILL;
  }
  refinement {
    AZI_FR = AZIMUTH;
    AZI_IN : RNGOUT;
    AZI_OUT : SAR_OUT;
    AZ_KERN : AZKERN;
    NFFT = NFFT_AZI;
    RNG_FFT =  NFFT_RNG;
  }
  actor RANGE {
    computation = "SUBGRAPH";
  }
  actor AZIMUTH {
    computation = "SUBGRAPH";
  }
}
```

DIF RANGE Processing Graph Generated from SPGN

```
dif RNG_FR
 {
 topology
  {
  nodes =
   PAD,
   WEIGHT,
   COMPRESS,
   COMPENSATE;
  edges =
   PADDED (PAD, WEIGHT),
   WEIGHTED (WEIGHT, COMPRESS),
   COMPRESSED (COMPRESS, COMPENSATE);
  }
 interface
  {
  inputs =
   RANGE_IN : PAD;
  outputs =
   RANGE_OUT : COMPENSATE;
  }
 parameter
  {
  NFFT = 2048;
  NR = 2032;
  NPAD = 16;
  TAYLOR_WTS;
  RCS_WTS;
  PAD_VAL = (0.00000000000000, 0.00000000000000);
  NRNG = 512;
  }
 production
  {
  PADDED = 1048576;
  WEIGHTED = 1048576;
  COMPRESSED = 1048576;
  RANGE_OUT = 1048576;
  }
 consumption
  {
  RANGE_IN = 1040384;
  PADDED = 1048576;
  WEIGHTED = 1048576;
  COMPRESSED = 1048576;
  }
 attribute threshold
  {
  RANGE_IN = 1040384;
  PADDED = 1048576;
  WEIGHTED = 1048576;
  COMPRESSED = 1048576;
  }
 actor PAD
```

```
  {
  computation = "D_VFILL";
  N = NR;
  P = NPAD;
  J = "UNUSED";
  V = PAD_VAL;
  X = RANGE_IN;
  Y = PADDED;
  }
actor WEIGHT
  {
  computation = "D_VMUL";
  N = NFFT;
  F = "UNUSED";
  X = PADDED;
  Y = TAYLOR_WTS;
  Z = WEIGHTED;
  }
actor COMPRESS
  {
  computation = "D_FFT";
  N = NFFT;
  M = "UNUSED";
  FI = "UNUSED";
  B = "UNUSED";
  OV = "UNUSED";
  X = WEIGHTED;
  Y = COMPRESSED;
  }
actor COMPENSATE
  {
  computation = "D_VMUL";
  N = NFFT;
  F = "UNUSED";
  X = COMPRESSED;
  Y = RCS_WTS;
  Z = RANGE_OUT;
  }
}
```

DIF AZIMUTH Processing Graph Generated from SPGN

```
dif AZI_FR
  {
  topology
    {
    nodes =
      CORNERTURN,
      FFT,
      CONVL,
      IFFT;
    edges =
      YFCO (CORNERTURN, FFT),
      Y_AZ (FFT, CONVL),
      VMAUL (CONVL, IFFT);
    }
  interface
    {
    inputs =
      AZI_N : CORNERTURN;
    outputs =
      AZI_OUT : IFFT;
    }
  parameter
    {
    NFFT = 1024;
    RNG_FFT = 2048;
    AZ_KERN;
    }
  production
    {
    YFCO = 2097152;
    Y_AZ = 2097152;
    VMAUL = 2097152;
    AZI_OUT = 1048576;
    }
  consumption
    {
    AZI_N = 1048576;
    YFCO = 2097152;
    Y_AZ = 2097152;
    VMAUL = 2097152;
    }
  attribute threshold
    {
    AZI_N = 2097152;
    YFCO = 2097152;
    Y_AZ = 2097152;
    VMAUL = 2097152;
    }
  actor CORNERTURN
    {
    computation = "D_MTRANS";
    M = NFFT;
    N = RNG_FFT;
    X = AZI_N;
  Y = YFCO;
  }
actor FFT
  {
  computation = "D_FFT";
  N = NFFT;
  M = "UNUSED";
  FI = "UNUSED";
  B = "UNUSED";
  OV = "UNUSED";
  X = YFCO;
  Y = Y_AZ;
  }
actor CONVL
  {
  computation = "D_VMUL";
  N = NFFT;
  F = "UNUSED";
  X = Y_AZ;
  Y = AZ_KERN;
  Z = VMAUL;
  }
actor IFFT
  {
  computation = "D_FFT";
  N = NFFT;
  M = "(NFFT / 2)";
  FI = 1;
  B = "((NFFT / 2) + 1)";
  OV = "UNUSED";
  X = VMAUL;
  Y = AZI_OUT;
  }
}
```

DIF SAR Graph

```
dif FR_SAR {
 topology {
  nodes = RANGE, AZIMUTH;
  edges = RNGOUT (RANGE, AZIMUTH);
 }
 interface {
  inputs = SAR_IN : RANGE,
         TAYLOR : RANGE,
         RCS : RANGE,
         AZKERN : AZIMUTH;
  outputs = SAR_OUT : AZIMUTH;
 }
 parameter {
  NFFT_RNG = 2048;
  NFFT_AZI = 1024;
  N_R = 2032;
  NFILL = 16;
 }
 refinement {
  RNG_FR = RANGE;
  RANGE_IN : SAR_IN;
  RANGE_OUT : RNGOUT;
  TAYLOR_WTS : TAYLOR;
  RCS_WTS : RCS;
  NFFT =  NFFT_RNG;
  NR = N_R;
  NPAD = NFILL;
 }
 refinement {
  AZI_FR = AZIMUTH;
  AZI_IN : RNGOUT;
  AZI_OUT : SAR_OUT;
  AZ_KERN : AZKERN;
  NFFT = NFFT_AZI;
  RNG_FFT =  NFFT_RNG;
 }
 actor RANGE {
  computation = "SUBGRAPH";
 }
 actor AZIMUTH {
  computation = "SUBGRAPH";
 }
}
```

DIF RANGE Graph

```
dif RNG_FR {
 topology {
  nodes = PAD, WEIGHT, COMPRESS, COMPENSATE;
  edges = PADDED (PAD, WEIGHT),
        WEIGHTED (WEIGHT, COMPRESS),
        COMPRESSED (COMPRESS, COMPENSATE);
 }
 interface {
  inputs = RANGE_IN : PAD,
         TAYLOR_WTS : WEIGHT,
         RCS_WTS : COMPENSATE;
  outputs = RANGE_OUT : COMPENSATE;
 }
 parameter {
  NFFT;
  NR;
  NPAD;
  NRNG = 512;
  PAD_VAL = 0.0;
 }
 actor PAD {
  computation = "D_VFILL" ;
  N = NR;
  P = NPAD;
  V = PAD_VAL;
  X = RANGE_IN;
  Y = PADDED;
 }
 actor WEIGHT {
  computation = "D_VMUL";
  N = NFFT;
  X = PADDED;
  Y = TAYLOR_WTS;
  Z = WEIGHTED;
 }
 actor COMPRESS {
  computation = "D_FFT";
  N = NFFT;
  FI = 0;
  X = WEIGHTED;
  Y = COMPRESSED;
 }
 actor COMPENSATE {
  computation = "D_VMUL";
  N = NFFT;
  X = COMPRESSED;
  Y = RCS_WTS;
  Z = RANGE_OUT;
 }
}
```

DIF AZIMUTH Graph

```
dif AZI_FR {
 topology {
  nodes = CORNERTURN, FFT, CONVL, IFFT;
  edges = YFCO (CORNERTURN, FFT),
       Y_AZ (FFT, CONVL),
       VMAUL (CONVL, IFFT);
 }
 interface {
  inputs = AZI_IN : CORNERTURN,
        AZ_KERN : CONVL;
  outputs = AZI_OUT : IFFT;
 }
 parameter {
  NFFT;
  RNG_FFT;
 }
 actor CORNERTURN {
  computation = "D_MTRAN";
  M = NFFT;
  N = RNG_FFT;
  X = AZI_IN;
  Y = YFCO;
 }
 actor FFT {
  computation = "D_FFT";
  N = NFFT;
  FI = 0;
  X = YFCO;
  Y = Y_AZ;
 }
 actor CONVL {
  computation = "D_VMUL";
  N = NFFT;
  X = Y_AZ;
  Y = AZ_KERN;
  Z = VMAUL;
 }
 actor IFFT {
  computation = "D_FFT";
  N = NFFT;
  FI = 1;
  X = VMAUL;
  Y = AZI_OUT;
 }
}
```

SAR Imported SPGN with Manual Edits

```
%GRAPH (FR_SAR
  INPUTQ =
    SAR_IN : CFLOAT,

    TAYLOR : CFLOAT ARRAY(2048),  %% these 2 items were GIPs in source SPGN
    RCS : FLOAT ARRAY(2048),      %%  -> changed to Qs due to limitations
                         %%    of DIF "interface" and "params"
                         %%    statements?

    AZKERN : CFLOAT ARRAY(1024)   %% was a VAR in source SPGN
                         %%  -> also changed for <above> reason?

  OUTPUTQ = SAR_OUT : CFLOAT
  )

%GIP (NFFT_AZI : INT
  INITIALIZE TO 1024
  )
%GIP (NFFT_RNG : INT
  INITIALIZE TO 2048
  )
%GIP (NFILL : INT
  INITIALIZE TO 16
  )
%GIP (N_R : INT
  INITIALIZE TO 2032
  )

%QUEUE (RNGOUT : CFLOAT
  INITIALIZE TO                        %% initialization (amt & vals)
  (NFFT_RNG*NFFT_AZI)/2 OF <0.0E0,0.0E0>)   %% lost in DIF representation

%SUBGRAPH (RANGE
  GRAPH = RNG_FR
  GIP =
    NFFT_RNG,
    NFILL,
    N_R
  INPUTQ =
    SAR_IN,
    TAYLOR,
    RCS
  OUTPUTQ =
    RNGOUT
  )
%SUBGRAPH (AZIMUTH
  GRAPH = AZI_FR
  GIP =
    NFFT_AZI,
    NFFT_RNG
  INPUTQ =
    RNGOUT,
    AZKERN
  OUTPUTQ =
    SAR_OUT
  )
%ENDGRAPH
```

RANGE Processing Imported SPGN with Manual Edits

```
%GRAPH (RNG_FR
  GIP =
    NFFT : INT,
    NPAD : INT,
    NR : INT
  INPUTQ =
    RANGE_IN : CFLOAT,
    TAYLOR_WTS : CFLOAT ARRAY(2048),
    RCS_WTS : FLOAT ARRAY(2048)
  OUTPUTQ = RANGE_OUT : CFLOAT
  )

%GIP (NRNG : INT
  INITIALIZE TO 512
  )

%GIP (PAD_VAL : CFLOAT          %% PAD_VAL should be CFLOAT
  INITIALIZE TO <0.0E0, 0.0E0>
  )

%QUEUE (PADDED : CFLOAT)
%QUEUE (WEIGHTED : CFLOAT)
%QUEUE (COMPRESSED : CFLOAT)

%NODE (PAD
  PRIMITIVE = D_VFILL
  PRIM_IN =
    NR,
    NPAD,
    UNUSED,                %% UNUSED param missing in DIF
    PAD_VAL,
    RANGE_IN
      THRESHOLD = NRNG*NR     %% no NEPS in DIF
  PRIM_OUT =
    PADDED
  )
%NODE (WEIGHT
  PRIMITIVE = D_VMUL
  PRIM_IN =
    NFFT,
    UNUSED,                %% UNUSED params missing in DIF
    PADDED
      THRESHOLD = NRNG*NFFT,   %% no NEPS in DIF
    TAYLOR_WTS
      THRESHOLD = 1          %% TAYLOR_WTS were specified as a GIP,
      CONSUME = 0           %%  not a Q, in source SPGN
  PRIM_OUT =
    WEIGHTED
  )
%NODE (COMPRESS
  PRIMITIVE = D_FFT
  PRIM_IN =
    NFFT,
    UNUSED,                %% UNUSED params missing in DIF
    0,
    UNUSED,
    UNUSED,
    WEIGHTED
      THRESHOLD = NRNG*NFFT
  PRIM_OUT =
    COMPRESSED
  )
%NODE (COMPENSATE
  PRIMITIVE = D_VMUL
  PRIM_IN =
    NFFT,
    UNUSED,
    COMPRESSED
      THRESHOLD = NRNG*NFFT,
    RCS_WTS
      THRESHOLD = 1          %% RCS_WTS were specified as a GIP,
      CONSUME = 0           %%  not a Q, in source SPGN
  PRIM_OUT =
    RANGE_OUT
  )
%ENDGRAPH
```

The DIF Specification of the Multi-rate Filter Bank Application

```
dif Truncated_Sinewave {
    topology {
        nodes = Ramp,
                Pulse,
                TrigFunction,
                MultiplyDivide,
                ${center/2}$;
        edges = e0 (Ramp, TrigFunction),
                e1 (Pulse, ${center/2}$),
                e2 (TrigFunction, MultiplyDivide),
                e3 (${center/2}$, MultiplyDivide);
    }
    interface {
        outputs = output:MultiplyDivide;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute frequency { = 0.6283185307179586; }
    attribute center { = 50; }
    attribute lengthOfSineBurst { = 50; }
    actor Ramp {
        computation = "ptolemy.actor.lib.Ramp";
        firingCountLimit : PARAMETER = 0;
        init : PARAMETER = -79.57747154594767;
        step : PARAMETER = 0.6283185307179586;
        output : OUTPUT = e0;
    }
    actor Pulse {
        computation = "ptolemy.actor.lib.Pulse";
        firingCountLimit : PARAMETER = 0;
        indexes : PARAMETER =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,5
0};
        values : PARAMETER =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
        repeat : PARAMETER = false;
        output : OUTPUT = e1;
    }
    actor TrigFunction {
        computation = "ptolemy.actor.lib.TrigFunction";
        function : PARAMETER = "cos";
        input : INPUT = e0;
        output : OUTPUT = e2;
    }
    actor MultiplyDivide {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = e2, e3;
        output : OUTPUT = output;
    }
    actor ${center/2}$ {
        computation = "mapss.applications.lib.SampleDelay";
        initialOutputs : PARAMETER = {0};
        repetitionCount : PARAMETER = 25;
        repetitionValue : PARAMETER = 0;
        input : INPUT = e1;
        output : OUTPUT = e3;
    }
}

dif Analysis_Filter_Pair1 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork0;
        edges = e4 (fork0, FIR_highpass),
                e5 (fork0, FIR_lowpass);
    }
    interface {
        inputs = input:fork0;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e4;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e5;
        output : OUTPUT = output2;
    }
    actor fork0 {
        computation = "dif.fork";
    }
}

dif Analysis_Filter_Pair2 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork1;
        edges = e6 (fork1, FIR_highpass),
                e7 (fork1, FIR_lowpass);
    }
    interface {
        inputs = input:fork1;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e6;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e7;
        output : OUTPUT = output2;
    }
    actor fork1 {
        computation = "dif.fork";
    }
}

dif Analysis_Filter_Pair3 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork2;
        edges = e8 (fork2, FIR_highpass),
                e9 (fork2, FIR_lowpass);
    }
    interface {
        inputs = input:fork2;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e8;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e9;
        output : OUTPUT = output2;
    }
    actor fork2 {
        computation = "dif.fork";
    }
}

dif Analysis_Filter_Pair4 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork3;
        edges = e10 (fork3, FIR_highpass),
                e11 (fork3, FIR_lowpass);
    }
    interface {
        inputs = input:fork3;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e10;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e11;
        output : OUTPUT = output2;
    }
    actor fork3 {
        computation = "dif.fork";
    }
}

dif Analysis_Filter_Pair5 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork4;
        edges = e12 (fork4, FIR_highpass),
                e13 (fork4, FIR_lowpass);
    }
    interface {
        inputs = input:fork4;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e12;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e13;
        output : OUTPUT = output2;
    }
    actor fork4 {
        computation = "dif.fork";
    }
}

dif Analysis_Filter_Pair6 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork5;
        edges = e14 (fork5, FIR_highpass),
                e15 (fork5, FIR_lowpass);
    }
    interface {
        inputs = input:fork5;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e14;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e15;
        output : OUTPUT = output2;
    }
    actor fork5 {
        computation = "dif.fork";
    }
}
```

```
dif Analysis_Filter_Pair7 {
    topology {
        nodes = FIR_highpass,
                FIR_lowpass,
                fork6;
        edges = e16 (fork6, FIR_highpass),
                e17 (fork6, FIR_lowpass);
    }
    interface {
        inputs = input:fork6;
        outputs = output1:FIR_highpass,
                  output2:FIR_lowpass;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [211,474]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-7.0E-4,-
0.011344,0.011408,0.023464,-0.001747,-0.044403,-0.204294,0.647669,-
0.647669,0.204294,0.044403,0.001747,-0.023464,-0.011408,0.011344,7.0E-
4,-0.001224};
        input : INPUT = e16;
        output : OUTPUT = output1;
    }
    actor FIR_lowpass {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 2;
        decimationPhase : PARAMETER = 1;
        interpolation : PARAMETER = 1;
        taps : PARAMETER = {0.001224,-6.98E-4,-
0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224};
        input : INPUT = e17;
        output : OUTPUT = output2;
    }
    actor fork6 {
        computation = "dif.fork";
    }
}

dif Reconstruction_Filter_Pair1 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e18 (FIR_highpass_R, AddSubtract),
                e19 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
            outputs = output:AddSubtract;
        }
        attribute _vergilSize { = [600,400]; }
        attribute _vergilLocation { = [232,252]; }
        attribute lowpass { = "qmf.lowpass.filter"; }
        attribute highpass { = "qmf.highpass.filter"; }
        actor FIR_highpass_R {
            computation = "ptolemy.domains.sdf.lib.FIR";
            decimation : PARAMETER = 1;
            decimationPhase : PARAMETER = 0;
            interpolation : PARAMETER = 2;
            taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
            input : INPUT = input1;
            output : OUTPUT = e18;
        }
        actor FIR_lowpass_R {
            computation = "ptolemy.domains.sdf.lib.FIR";
            decimation : PARAMETER = 1;
            decimationPhase : PARAMETER = 0;
            interpolation : PARAMETER = 2;
            taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
            input : INPUT = input2;
            output : OUTPUT = e19;
        }
        actor AddSubtract {
            computation = "ptolemy.actor.lib.AddSubtract";
            plus : INPUT = e18, e19;
            output : OUTPUT = output;
        }
}

dif Reconstruction_Filter_Pair2 {
        topology {
            nodes = FIR_highpass_R,
                    FIR_lowpass_R,
                    AddSubtract;
            edges = e20 (FIR_highpass_R, AddSubtract),
                    e21 (FIR_lowpass_R, AddSubtract);
        }
        interface {
            inputs = input1:FIR_highpass_R,
                     input2:FIR_lowpass_R;
            outputs = output:AddSubtract;
        }
        attribute _vergilSize { = [600,400]; }
        attribute _vergilLocation { = [232,252]; }
        attribute lowpass { = "qmf.lowpass.filter"; }
        attribute highpass { = "qmf.highpass.filter"; }
        actor FIR_highpass_R {
            computation = "ptolemy.domains.sdf.lib.FIR";
            decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e20;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e21;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e20, e21;
        output : OUTPUT = output;
    }
}

dif Reconstruction_Filter_Pair3 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e22 (FIR_highpass_R, AddSubtract),
                e23 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e22;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e23;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e22, e23;
        output : OUTPUT = output;
    }
}

dif Reconstruction_Filter_Pair4 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e24 (FIR_highpass_R, AddSubtract),
                e25 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e24;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e25;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e24, e25;
        output : OUTPUT = output;
    }
}

dif Reconstruction_Filter_Pair5 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e26 (FIR_highpass_R, AddSubtract),
                e27 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e26;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e27;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e26, e27;
        output : OUTPUT = output;
    }
}

dif Reconstruction_Filter_Pair6 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e28 (FIR_highpass_R, AddSubtract),
                e29 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e28;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e29;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e28, e29;
        output : OUTPUT = output;
    }
}

dif Reconstruction_Filter_Pair7 {
    topology {
        nodes = FIR_highpass_R,
                FIR_lowpass_R,
                AddSubtract;
        edges = e30 (FIR_highpass_R, AddSubtract),
                e31 (FIR_lowpass_R, AddSubtract);
    }
    interface {
        inputs = input1:FIR_highpass_R,
                 input2:FIR_lowpass_R;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [232,252]; }
    attribute lowpass { = "qmf.lowpass.filter"; }
    attribute highpass { = "qmf.highpass.filter"; }
    actor FIR_highpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {-0.001224,-6.98E-4,0.011833,0.011682,-
0.071283,-0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224};
        input : INPUT = input1;
        output : OUTPUT = e30;
    }
    actor FIR_lowpass_R {
        computation = "ptolemy.domains.sdf.lib.FIR";
        decimation : PARAMETER = 1;
        decimationPhase : PARAMETER = 0;
        interpolation : PARAMETER = 2;
        taps : PARAMETER = {0.001224,7.0E-4,-0.011344,-
0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224};
        input : INPUT = input2;
        output : OUTPUT = e31;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e30, e31;
        output : OUTPUT = output;
    }
}

dif PreScaler {
    topology {
        nodes = Scale,
                Limiter,
                AddSubtract,
                Const;
        edges = e32 (Scale, Limiter),
                e33 (Limiter, AddSubtract),
                e34 (Const, AddSubtract);
    }
    interface {
        inputs = input:Scale;
        outputs = output:AddSubtract;
    }
    attribute _vergilSize { = [600,400]; }
    attribute _vergilLocation { = [181,466]; }
    attribute gain { = 100; }
    attribute offset { = 128; }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        factor : PARAMETER = 100;
        scaleOnLeft : PARAMETER = true;
        input : INPUT = input;
        output : OUTPUT = e32;
    }
    actor Limiter {
        computation = "ptolemy.actor.lib.Limiter";
        bottom : PARAMETER = -128.0;
        top : PARAMETER = 127.0;
        input : INPUT = e32;
        output : OUTPUT = e33;
    }
    actor AddSubtract {
        computation = "ptolemy.actor.lib.AddSubtract";
        plus : INPUT = e33, e34;
        output : OUTPUT = output;
    }
    actor Const {
        computation = "ptolemy.actor.lib.Const";
        value : PARAMETER = 128;
        output : OUTPUT = e34;
    }
}

dif FilterBank {
    topology {
        nodes = Truncated_Sinewave,
                SequencePlotter,
                Analysis_Filter_Pair1,
                Analysis_Filter_Pair2,
                Analysis_Filter_Pair3,
                Analysis_Filter_Pair4,
                Analysis_Filter_Pair5,
                Analysis_Filter_Pair6,
                Analysis_Filter_Pair7,
                Reconstruction_Filter_Pair1,
                $Repeat{2,1}$,
                Reconstruction_Filter_Pair2,
                Reconstruction_Filter_Pair3,
                $Repeat{4,1}$,
                $Repeat{8,1}$,
                $Repeat{16,1}$,
                $Repeat{32,1}$,
                Reconstruction_Filter_Pair4,
                Reconstruction_Filter_Pair5,
                Reconstruction_Filter_Pair6,
                Reconstruction_Filter_Pair7,
                $Repeat1{64,1}$,
                $Repeat2{64,1}$,
                Commutator,
                PreScaler,
                FileWriter,
                ${base*63/2}$,
                ${base*31/2}$,
                ${base*15/2}$,
                ${base*7/2}$,
                ${base*3/2}$,
                ${8}$,
                ${base*63*2 + base}$,
                ${base*63/2 + 1}$,
                    ${base*31/2 + 1}$,
                    ${base*15/2 + 1}$,
                    ${base*7/2 + 1}$,
                    ${base*3/2 + 1}$,
                    ${9}$,
                    fork7,
                    fork8,
                    fork9,
                    fork10,
                    fork11,
                    fork12,
                    fork13,
                    fork14,
                    fork15;
            edges = e35 (Truncated_Sinewave, fork7),
                    e36 (fork7, Analysis_Filter_Pair1),
                    e37 (fork7, ${base*63*2 + base}$),
                    e38 (Analysis_Filter_Pair1, ${base*63/2}$),
                    e39 (Analysis_Filter_Pair1, Analysis_Filter_Pair2),
                    e40 (Analysis_Filter_Pair2, ${base*31/2}$),
                    e41 (Analysis_Filter_Pair2, Analysis_Filter_Pair3),
                    e42 (Analysis_Filter_Pair3, ${base*15/2}$),
                    e43 (Analysis_Filter_Pair3, Analysis_Filter_Pair4),
                    e44 (Analysis_Filter_Pair4, ${base*7/2}$),
                    e45 (Analysis_Filter_Pair4, Analysis_Filter_Pair5),
                    e46 (Analysis_Filter_Pair5, ${base*3/2}$),
                    e47 (Analysis_Filter_Pair5, Analysis_Filter_Pair6),
                    e48 (Analysis_Filter_Pair6, ${8}$),
                    e49 (Analysis_Filter_Pair6, Analysis_Filter_Pair7),
                    e50 (Analysis_Filter_Pair7, fork8),
                    e51 (fork8, Reconstruction_Filter_Pair7),
                    e52 (fork8, $Repeat1{64,1}$),
                    e53 (Analysis_Filter_Pair7, fork9),
                    e54 (fork9, Reconstruction_Filter_Pair7),
                    e55 (fork9, $Repeat2{64,1}$),
                    e56 (Reconstruction_Filter_Pair1, SequencePlotter),
                    e57 ($Repeat{2,1}$, Commutator),
                    e58 (Reconstruction_Filter_Pair2,
Reconstruction_Filter_Pair1),
                    e59 (Reconstruction_Filter_Pair3,
Reconstruction_Filter_Pair2),
                    e60 ($Repeat{4,1}$, Commutator),
                    e61 ($Repeat{8,1}$, Commutator),
                    e62 ($Repeat{16,1}$, Commutator),
                    e63 ($Repeat{32,1}$, Commutator),
                    e64 (Reconstruction_Filter_Pair4,
Reconstruction_Filter_Pair3),
                    e65 (Reconstruction_Filter_Pair5,
Reconstruction_Filter_Pair4),
                    e66 (Reconstruction_Filter_Pair6,
Reconstruction_Filter_Pair5),
                    e67 (Reconstruction_Filter_Pair7,
Reconstruction_Filter_Pair6),
                    e68 ($Repeat1{64,1}$, Commutator),
                    e69 ($Repeat2{64,1}$, Commutator),
                    e70 (Commutator, PreScaler),
                    e71 (PreScaler, FileWriter),
                e72 (${base*63/2}$, fork10),
                e73 (fork10, Commutator),
                e74 (fork10, ${base*63/2 + 1}$),
                e75 (${base*31/2}$, fork11),
                e76 (fork11, $Repeat{2,1}$),
                e77 (fork11, ${base*31/2 + 1}$),
                e78 (${base*15/2}$, fork12),
                e79 (fork12, $Repeat{4,1}$),
                e80 (fork12, ${base*15/2 + 1}$),
                e81 (${base*7/2}$, fork13),
                e82 (fork13, $Repeat{8,1}$),
                e83 (fork13, ${base*7/2 + 1}$),
                e84 (${base*3/2}$, fork14),
                e85 (fork14, $Repeat{16,1}$),
                e86 (fork14, ${base*3/2 + 1}$),
                e87 (${8}$, fork15),
                e88 (fork15, $Repeat{32,1}$),
                e89 (fork15, ${9}$),
                e90 (${base*63*2 + base}$, SequencePlotter),
                e91 (${base*63/2 + 1}$, Reconstruction_Filter_Pair1),
                e92 (${base*31/2 + 1}$, Reconstruction_Filter_Pair2),
                e93 (${base*15/2 + 1}$, Reconstruction_Filter_Pair3),
                e94 (${base*7/2 + 1}$, Reconstruction_Filter_Pair4),
                e95 (${base*3/2 + 1}$, Reconstruction_Filter_Pair5),
                e96 (${9}$, Reconstruction_Filter_Pair6);
    }
    refinement {
        Truncated_Sinewave = Truncated_Sinewave;
        output : e35;
    }
    refinement {
        Analysis_Filter_Pair1 = Analysis_Filter_Pair1;
        input : e36;
        output1 : e38;
        output2 : e39;
    }
    refinement {
        Analysis_Filter_Pair2 = Analysis_Filter_Pair2;
        input : e39;
        output1 : e40;
        output2 : e41;
    }
    refinement {
        Analysis_Filter_Pair3 = Analysis_Filter_Pair3;
        input : e41;
        output1 : e42;
        output2 : e43;
    }
    refinement {
        Analysis_Filter_Pair4 = Analysis_Filter_Pair4;
        input : e43;
        output1 : e44;
        output2 : e45;
    }
    refinement {
        Analysis_Filter_Pair5 = Analysis_Filter_Pair5;
        input : e45;
        output1 : e46;
        output2 : e47;
    }
    refinement {
        Analysis_Filter_Pair6 = Analysis_Filter_Pair6;
        input : e47;
        output1 : e48;
        output2 : e49;
    }
    refinement {
        Analysis_Filter_Pair7 = Analysis_Filter_Pair7;
        input : e49;
        output1 : e50;
        output2 : e53;
    }
    refinement {
        Reconstruction_Filter_Pair1 = Reconstruction_Filter_Pair1;
        input1 : e91;
        input2 : e58;
        output : e56;
    }
    refinement {
        Reconstruction_Filter_Pair2 = Reconstruction_Filter_Pair2;
        input1 : e92;
        input2 : e59;
        output : e58;
    }
    refinement {
        Reconstruction_Filter_Pair3 = Reconstruction_Filter_Pair3;
        input1 : e93;
        input2 : e64;
        output : e59;
    }
    refinement {
        Reconstruction_Filter_Pair4 = Reconstruction_Filter_Pair4;
        input1 : e94;
        input2 : e65;
        output : e64;
    }
    refinement {
        Reconstruction_Filter_Pair5 = Reconstruction_Filter_Pair5;
        input1 : e95;
        input2 : e66;
        output : e65;
    }
    refinement {
        Reconstruction_Filter_Pair6 = Reconstruction_Filter_Pair6;
        input1 : e96;
        input2 : e67;
        output : e66;
    }
    refinement {
        Reconstruction_Filter_Pair7 = Reconstruction_Filter_Pair7;
        input1 : e51;
        input2 : e54;
        output : e67;
    }
    refinement {
        PreScaler = PreScaler;
        input : e70;
        output : e71;
    }
    attribute _vergilSize { = [1073,879]; }
    attribute _vergilLocation { = [-4,12]; }
    attribute base { = 17; }
    attribute noIterations { = 1600; }
    attribute _windowProperties { = "{bounds = {3, 10, 1288, 988},
maximized = false}"; }
    attribute _vergilZoomFactor { = 1.0; }
    attribute _vergilCenter { = {536.5,439.5}; }
    actor Truncated_Sinewave {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor SequencePlotter {
        computation = "ptolemy.actor.lib.gui.SequencePlotter";
        fillOnWrapup : PARAMETER = true;
        _windowProperties : PARAMETER = "{bounds = {386, 332, 508,
359}}";
        _plotSize : PARAMETER = [500,300];
        startingDataset : PARAMETER = 0;
        xInit : PARAMETER = 0.0;
        xUnit : PARAMETER = 1.0;
        input : INPUT = e56, e90;
    }
    actor Analysis_Filter_Pair1 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair2 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair3 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair4 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair5 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair6 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Analysis_Filter_Pair7 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor Reconstruction_Filter_Pair1 {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor $Repeat{2,1}$ {
        computation = "ptolemy.domains.sdf.lib.Repeat";
        numberOfTimes : PARAMETER = 2;
        blockSize : PARAMETER = 1;
        input : INPUT = e76;
        output : OUTPUT = e57;
```

```
}
actor Reconstruction_Filter_Pair2 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor Reconstruction_Filter_Pair3 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor $Repeat{4,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 4;
    blockSize : PARAMETER = 1;
    input : INPUT = e79;
    output : OUTPUT = e60;
}
actor $Repeat{8,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 8;
    blockSize : PARAMETER = 1;
    input : INPUT = e82;
    output : OUTPUT = e61;
}
actor $Repeat{16,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 16;
    blockSize : PARAMETER = 1;
    input : INPUT = e85;
    output : OUTPUT = e62;
}
actor $Repeat{32,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 32;
    blockSize : PARAMETER = 1;
    input : INPUT = e88;
    output : OUTPUT = e63;
}
actor Reconstruction_Filter_Pair4 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor Reconstruction_Filter_Pair5 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor Reconstruction_Filter_Pair6 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor Reconstruction_Filter_Pair7 {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor $Repeat1{64,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 64;
    blockSize : PARAMETER = 1;
    input : INPUT = e52;
    output : OUTPUT = e68;
}
actor $Repeat2{64,1}$ {
    computation = "ptolemy.domains.sdf.lib.Repeat";
    numberOfTimes : PARAMETER = 64;
    blockSize : PARAMETER = 1;
    input : INPUT = e55;
    output : OUTPUT = e69;
}
actor Commutator {
    computation = "ptolemy.actor.lib.Commutator";
    input : INPUT = e57, e60, e61, e62, e63, e68, e69, e73;
    output : OUTPUT = e70;
}
actor PreScaler {
    computation = "ptolemy.actor.TypedCompositeActor";
}
actor FileWriter {
    computation = "ptolemy.actor.lib.FileWriter";
    filename : PARAMETER = "c:\\testout.4";
    input : INPUT = e71;
}
actor ${base*63/2}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 535;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e38;
    output : OUTPUT = e72;
}
actor ${base*31/2}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 263;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e40;
    output : OUTPUT = e75;
}
actor ${base*15/2}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 127;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e42;
    output : OUTPUT = e78;
}
actor ${base*7/2}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 59;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e44;
    output : OUTPUT = e81;
}
actor ${base*3/2}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 25;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e46;
    output : OUTPUT = e84;
}
```

```
actor ${8}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 8;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e48;
    output : OUTPUT = e87;
}
actor ${base*63*2 + base}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 2159;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e37;
    output : OUTPUT = e90;
}
actor ${base*63/2 + 1}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 536;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e74;
    output : OUTPUT = e91;
}
actor ${base*31/2 + 1}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 264;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e77;
    output : OUTPUT = e92;
}
actor ${base*15/2 + 1}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 128;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e80;
    output : OUTPUT = e93;
}
actor ${base*7/2 + 1}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 60;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e83;
    output : OUTPUT = e94;
}
actor ${base*3/2 + 1}$ {
    computation = "mapss.applications.lib.SampleDelay";
    initialOutputs : PARAMETER = {0};
    repetitionCount : PARAMETER = 26;
    repetitionValue : PARAMETER = 0;
    input : INPUT = e86;
    output : OUTPUT = e95;
}
actor ${9}$ {
        computation = "mapss.applications.lib.SampleDelay";
        initialOutputs : PARAMETER = {0};
        repetitionCount : PARAMETER = 9;
        repetitionValue : PARAMETER = 0;
        input : INPUT = e89;
        output : OUTPUT = e96;
    }
    actor fork7 {
        computation = "dif.fork";
    }
    actor fork8 {
        computation = "dif.fork";
    }
    actor fork9 {
        computation = "dif.fork";
    }
    actor fork10 {
        computation = "dif.fork";
    }
    actor fork11 {
        computation = "dif.fork";
    }
    actor fork12 {
        computation = "dif.fork";
    }
    actor fork13 {
        computation = "dif.fork";
    }
    actor fork14 {
        computation = "dif.fork";
    }
    actor fork15 {
        computation = "dif.fork";
    }
}
```

Multi Rate Filter SPGN (Top Level Graph)

```
%GRAPH (FilterBank    )

%GIP (_vergilCenter : FLOAT ARRAY(2)
   INITIALIZE TO {536.5,439.5}
   )
%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {-4,12}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {1073,879}
   )
%GIP (_vergilZoomFactor : FLOAT
   INITIALIZE TO 1.0
   )
%GIP (_windowProperties : <UNKNOWN>
   INITIALIZE TO {bounds = {3, 10, 1288, 988}, maximized = false}
   )
%GIP (base : INT
   INITIALIZE TO 17
   )
%GIP (noIterations : INT
   INITIALIZE TO 1600
   )

%QUEUE (e35)
%QUEUE (e36)
%QUEUE (e37)
%QUEUE (e38)
%QUEUE (e39)
%QUEUE (e40)
%QUEUE (e41)
%QUEUE (e42)
%QUEUE (e43)
%QUEUE (e44)
%QUEUE (e45)
%QUEUE (e46)
%QUEUE (e47)
%QUEUE (e48)
%QUEUE (e49)
%QUEUE (e50)
%QUEUE (e51)
%QUEUE (e52)
%QUEUE (e53)
%QUEUE (e54)
%QUEUE (e55)
%QUEUE (e56)
%QUEUE (e57)
%QUEUE (e58)
%QUEUE (e59)
%QUEUE (e60)
%QUEUE (e61)
%QUEUE (e62)
%QUEUE (e63)
%QUEUE (e64)
```

```
%QUEUE (e65)
%QUEUE (e66)
%QUEUE (e67)
%QUEUE (e68)
%QUEUE (e69)
%QUEUE (e70)
%QUEUE (e71)
%QUEUE (e72)
%QUEUE (e73)
%QUEUE (e74)
%QUEUE (e75)
%QUEUE (e76)
%QUEUE (e77)
%QUEUE (e78)
%QUEUE (e79)
%QUEUE (e80)
%QUEUE (e81)
%QUEUE (e82)
%QUEUE (e83)
%QUEUE (e84)
%QUEUE (e85)
%QUEUE (e86)
%QUEUE (e87)
%QUEUE (e88)
%QUEUE (e89)
%QUEUE (e90)
%QUEUE (e91)
%QUEUE (e92)
%QUEUE (e93)
%QUEUE (e94)
%QUEUE (e95)
%QUEUE (e96)

%SUBGRAPH (Truncated_Sinewave
   GRAPH = Truncated_Sinewave
   GIP =
   INPUTQ =
   OUTPUTQ =
      output = e35
   )
%SUBGRAPH (Analysis_Filter_Pair1
   GRAPH = Analysis_Filter_Pair1
   GIP =
   INPUTQ =
      input = e36
   OUTPUTQ =
      output1 = e38
      output2 = e39
   )
%SUBGRAPH (Analysis_Filter_Pair2
   GRAPH = Analysis_Filter_Pair2
   GIP =
   INPUTQ =
      input = e39
   OUTPUTQ =
      output1 = e40
      output2 = e41
   )
%SUBGRAPH (Analysis_Filter_Pair3
   GRAPH = Analysis_Filter_Pair3
   GIP =
   INPUTQ =
      input = e41
   OUTPUTQ =
      output1 = e42
      output2 = e43
   )
%SUBGRAPH (Analysis_Filter_Pair4
   GRAPH = Analysis_Filter_Pair4
   GIP =
   INPUTQ =
      input = e43
   OUTPUTQ =
      output1 = e44
      output2 = e45
   )
%SUBGRAPH (Analysis_Filter_Pair5
   GRAPH = Analysis_Filter_Pair5
   GIP =
   INPUTQ =
      input = e45
   OUTPUTQ =
      output1 = e46
      output2 = e47
   )
%SUBGRAPH (Analysis_Filter_Pair6
   GRAPH = Analysis_Filter_Pair6
   GIP =
   INPUTQ =
      input = e47
   OUTPUTQ =
      output1 = e48
      output2 = e49
   )
%SUBGRAPH (Analysis_Filter_Pair7
   GRAPH = Analysis_Filter_Pair7
   GIP =
   INPUTQ =
      input = e49
   OUTPUTQ =
      output1 = e50
      output2 = e53
   )
%SUBGRAPH (Reconstruction_Filter_Pair1
   GRAPH = Reconstruction_Filter_Pair1
   GIP =
   INPUTQ =
      input1 = e91
      input2 = e58
   OUTPUTQ =
      output = e56
   )
%SUBGRAPH (Reconstruction_Filter_Pair2
   GRAPH = Reconstruction_Filter_Pair2
   GIP =
   INPUTQ =
      input1 = e92
      input2 = e59
   OUTPUTQ =
      output = e58
   )
%SUBGRAPH (Reconstruction_Filter_Pair3
   GRAPH = Reconstruction_Filter_Pair3
   GIP =
   INPUTQ =
      input1 = e93
      input2 = e64
   OUTPUTQ =
      output = e59
   )
%SUBGRAPH (Reconstruction_Filter_Pair4
   GRAPH = Reconstruction_Filter_Pair4
   GIP =
   INPUTQ =
      input1 = e94
      input2 = e65
   OUTPUTQ =
      output = e64
   )
%SUBGRAPH (Reconstruction_Filter_Pair5
   GRAPH = Reconstruction_Filter_Pair5
   GIP =
   INPUTQ =
      input1 = e95
      input2 = e66
   OUTPUTQ =
      output = e65
   )
%SUBGRAPH (Reconstruction_Filter_Pair6
   GRAPH = Reconstruction_Filter_Pair6
   GIP =
   INPUTQ =
      input1 = e96
      input2 = e67
   OUTPUTQ =
      output = e66
   )
%SUBGRAPH (Reconstruction_Filter_Pair7
   GRAPH = Reconstruction_Filter_Pair7
   GIP =
   INPUTQ =
      input1 = e51
      input2 = e54
   OUTPUTQ =
      output = e67
   )
%SUBGRAPH (PreScaler
   GRAPH = PreScaler
   GIP =
   INPUTQ =
      input = e70
   OUTPUTQ =
      output = e71
   )
%ENDGRAPH
```

Analysis_Filter_Pair Subgraph SPGN

```
%GRAPH (Analysis_Filter_Pair1
   INPUTQ = input
   OUTPUTQ =
      output1
      output2
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {211,474}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (highpass : <UNKNOWN>
   INITIALIZE TO qmf.highpass.filter
   )
%GIP (lowpass : <UNKNOWN>
   INITIALIZE TO qmf.lowpass.filter
   )

%QUEUE (e4)
%QUEUE (e5)

%NODE (FIR_highpass
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 2
      decimationPhase = 1
      interpolation = 1
      taps = {0.001224,-7.0E-4,-0.011344,0.011408,0.023464,-0.001747,-
0.044403,-0.204294,0.647669,-0.647669,0.204294,0.044403,0.001747,-
0.023464,-0.011408,0.011344,7.0E-4,-0.001224}
      input = e4
      output = output1
   )
%NODE (FIR_lowpass
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 2
      decimationPhase = 1
      interpolation = 1
      taps = {0.001224,-6.98E-4,-0.011833,0.011682,0.071283,-0.030986,-
0.226242,0.069248,0.731574,0.731574,0.069248,-0.226242,-
0.030986,0.071283,0.011682,-0.011833,-6.98E-4,0.001224}
      input = e5
      output = output2
   )
%NODE (fork0
   PRIMITIVE = ptolemy.actor.TypedIORelation
   PRIM_IN =
   PRIM_OUT =
   )
```

```
%ENDGRAPH
```

PreScaler Subgraph SPGN

```
%GRAPH (PreScaler
   INPUTQ = input
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {181,466}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (gain : INT
   INITIALIZE TO 100
   )
%GIP (offset : INT
   INITIALIZE TO 128
   )

%QUEUE (e32)
%QUEUE (e33)
%QUEUE (e34)

%NODE (Scale
   PRIMITIVE = ptolemy.actor.lib.Scale
   PRIM_IN =
   PRIM_OUT =
      factor = 100
      scaleOnLeft = true
      input = input
      output = e32
   )
%NODE (Limiter
   PRIMITIVE = ptolemy.actor.lib.Limiter
   PRIM_IN =
   PRIM_OUT =
      bottom = -128.0
      top = 127.0
      input = e32
      output = e33
   )
%NODE (AddSubtract
   PRIMITIVE = ptolemy.actor.lib.AddSubtract
   PRIM_IN =
   PRIM_OUT =
      output = output
   )
%NODE (Const
   PRIMITIVE = ptolemy.actor.lib.Const
   PRIM_IN =
   PRIM_OUT =
      value = 128
      output = e34
   )
```

```
%ENDGRAPH
```

Reconstruction_Filter_Pair Subgraph SPGN

```
%GRAPH (Reconstruction_Filter_Pair1
   INPUTQ =
      input1
      input2
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {232,252}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (highpass : <UNKNOWN>
   INITIALIZE TO qmf.highpass.filter
   )
%GIP (lowpass : <UNKNOWN>
   INITIALIZE TO qmf.lowpass.filter
   )

%QUEUE (e18)
%QUEUE (e19)

%NODE (FIR_highpass_R
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 1
      decimationPhase = 0
      interpolation = 2
      taps = {-0.001224,-6.98E-4,0.011833,0.011682,-0.071283,-
0.030986,0.226242,0.069248,-0.731574,0.731574,-0.069248,-
0.226242,0.030986,0.071283,-0.011682,-0.011833,6.98E-4,0.001224}
      input = input1
      output = e18
   )
%NODE (FIR_lowpass_R
   PRIMITIVE = ptolemy.domains.sdf.lib.FIR
   PRIM_IN =
   PRIM_OUT =
      decimation = 1
      decimationPhase = 0
      interpolation = 2
      taps = {0.001224,7.0E-4,-0.011344,-0.011408,0.023464,0.001747,-
0.044403,0.204294,0.647669,0.647669,0.204294,-
0.044403,0.001747,0.023464,-0.011408,-0.011344,7.0E-4,0.001224}
      input = input2
      output = e19
   )
%NODE (AddSubtract
   PRIMITIVE = ptolemy.actor.lib.AddSubtract
   PRIM_IN =
   PRIM_OUT =
      output = output
   )
%ENDGRAPH
```

Truncated Sinewave Subgraph SPGN

```
%GRAPH (Truncated_Sinewave
   OUTPUTQ = output
   )

%GIP (_vergilLocation : INT ARRAY(2)
   INITIALIZE TO {232,252}
   )
%GIP (_vergilSize : INT ARRAY(2)
   INITIALIZE TO {600,400}
   )
%GIP (center : INT
   INITIALIZE TO 50
   )
%GIP (frequency : FLOAT
   INITIALIZE TO 0.6283185307179586
   )
%GIP (lengthOfSineBurst : INT
   INITIALIZE TO 50
   )

%QUEUE (e0)
%QUEUE (e1)
%QUEUE (e2)
%QUEUE (e3)

%NODE (Ramp
   PRIMITIVE = ptolemy.actor.lib.Ramp
   PRIM_IN =
   PRIM_OUT =
      firingCountLimit = 0
      init = -79.57747154594767
      step = 0.6283185307179586
      output = e0
   )
%NODE (Pulse
   PRIMITIVE = ptolemy.actor.lib.Pulse
   PRIM_IN =
   PRIM_OUT =
      firingCountLimit = 0
      indexes =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,5
0}
      values =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0}
      repeat = false
      output = e1
   )
%NODE (TrigFunction
   PRIMITIVE = ptolemy.actor.lib.TrigFunction
   PRIM_IN =
   PRIM_OUT =
      function = cos
      input = e0
      output = e2
   )
%NODE (MultiplyDivide
   PRIMITIVE = ptolemy.actor.lib.MultiplyDivide
   PRIM_IN =
   PRIM_OUT =
      output = output
   )
%NODE (_center_2__
   PRIMITIVE = mapss.applications.lib.SampleDelay
   PRIM_IN =
   PRIM_OUT =
      initialOutputs = {0}
      repetitionCount = 25
      repetitionValue = 0
      input = e1
      output = e3
   )
%ENDGRAPH
```

The DIF Specification of the Ported SAR in Ptolemy II

```
dif IFFT_SUBGRAPH {
    topology {
        nodes = IFFT,
                Scale,
                SequenceToArray,
                ArrayExtract,
                ArrayToSequence;
        edges = e1 (IFFT, Scale),
                e2 (Scale, SequenceToArray),
                e3 (SequenceToArray, ArrayExtract),
                e4 (ArrayExtract, ArrayToSequence);
    }
    interface {
        inputs = in:IFFT;
        outputs = out:ArrayToSequence;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order : PARAMETER = 7.0;
        input : INPUT = in;
        output : OUTPUT = e1;
    }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        input : INPUT = e1;
        output : OUTPUT = e2;
        factor : PARAMETER = 128;
    }
    actor SequenceToArray {
        computation = "ptolemy.domains.sdf.lib.SequenceToArray";
        input : INPUT = e2;
        output : OUTPUT = e3;
        arrayLength : PARAMETER = 128;
    }
    actor ArrayExtract {
        computation = "ptolemy.actor.lib.ArrayExtract";
        input : INPUT = e3;
        output : OUTPUT = e4;
        sourcePosition : PARAMETER = 64;
        extractLength : PARAMETER = 64;
        destinationPosition : PARAMETER = 0;
        outputArrayLength : PARAMETER = 64;
    }
    actor ArrayToSequence {
        computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
        input : INPUT = e4;
        output : OUTPUT = out;
        arrayLength : PARAMETER = 64;
    }
}

dif RNG_FR {
    topology {
        nodes = PAD,
                WEIGHT,
                COMPRESS,
                COMPENSATE;
        edges = PADDED (PAD, WEIGHT),
                WEIGHTED (WEIGHT, COMPRESS),
                COMPRESSED (COMPRESS, COMPENSATE);
    }
    interface {
        inputs = RANGE_IN:PAD,
                 TAYLOR_WTS:WEIGHT,
                 RCS_WTS:COMPENSATE;
        outputs = RANGE_OUT:COMPENSATE;
    }
    parameter {
        NFFT;
        NR;
        NPAD;
        PAD_VAL = (0.0,0.0);
    }
    actor PAD {
        computation = "mapss.applications.sar.SequencePad";
        inputLength : PARAMETER = NR;
        outputLength : PARAMETER = 256;
        padValue : PARAMETER = PAD_VAL;
        input : INPUT = RANGE_IN;
        output : OUTPUT = PADDED;
    }
    actor WEIGHT {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = PADDED, TAYLOR_WTS;
        output : OUTPUT = WEIGHTED;
    }
    actor COMPRESS {
        computation = "ptolemy.domains.sdf.lib.FFT";
        order : PARAMETER = 8.0;
        input : INPUT = WEIGHTED;
        output : OUTPUT = COMPRESSED;
    }
    actor COMPENSATE {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = COMPRESSED, RCS_WTS;
        output : OUTPUT = RANGE_OUT;
    }
}

dif AZI_FR {
    topology {
        nodes = CORNERTURN,
                FFT,
                CONVL,
                IFFT;
        edges = YFCO (CORNERTURN, FFT),
                Y_AZ (FFT, CONVL),
                VMAUL (CONVL, IFFT);
    }
    interface {
        inputs = AZI_N:CORNERTURN,
                 AZ_KERN:CONVL;
        outputs = AZI_OUT:IFFT;
    }
    parameter {
        NFFT;
        RNG_FFT;
    }
    refinement {
        IFFT_SUBGRAPH = IFFT;
        in : VMAUL;
        out : AZI_OUT;
    }
    actor CORNERTURN {
        computation = "mapss.applications.sar.MatrixTranspose";
        rowN : PARAMETER = NFFT;
        colN : PARAMETER = RNG_FFT;
        input : INPUT = AZI_N;
        output : OUTPUT = YFCO;
    }
    actor FFT {
        computation = "ptolemy.domains.sdf.lib.FFT";
        order : PARAMETER = 7.0;
        input : INPUT = YFCO;
        output : OUTPUT = Y_AZ;
    }
    actor CONVL {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = Y_AZ, AZ_KERN;
        output : OUTPUT = VMAUL;
    }
    actor IFFT {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
}

dif FR_SAR {
    topology {
        nodes = RANGE,
                AZIMUTH;
        edges = RNG_OUT (RANGE, AZIMUTH);
    }
    interface {
        inputs = SAR_IN:RANGE,
                 TAYLOR:RANGE,
                 RCS:RANGE,
                 AZ_KERN:AZIMUTH;
        outputs = SAR_OUT:AZIMUTH;
    }
    parameter {
        NFFT_RNG = 256;
        NFFT_AZI = 128;
        N_R = 235;
        NFILL = "NFFT_RNG-N_R";
    }
    refinement {
        RNG_FR = RANGE;
```

```
            RANGE_IN : SAR_IN;
            TAYLOR_WTS : TAYLOR;
            RCS_WTS : RCS;
            RANGE_OUT : RNG_OUT;
            NFFT = NFFT_RNG;
            NR = N_R;
            NPAD = NFILL;
        }
        refinement {
            AZI_FR = AZIMUTH;
            AZI_N : RNG_OUT;
            AZ_KERN : AZ_KERN;
            AZI_OUT : SAR_OUT;
            NFFT = NFFT_AZI;
            RNG_FFT = NFFT_RNG;
        }
        actor RANGE {
            computation = "ptolemy.actor.TypedCompositeActor";
        }
        actor AZIMUTH {
            computation = "ptolemy.actor.TypedCompositeActor";
        }
    }
```

Appendix C to Final Report on

DIF - A Language for Dataflow Graph Specification and Exchange

October 27, 2004

Sponsored by

Defense Advanced Research Projects Agency (DOD)
(Controlling DARPA Office)

ARPA Order C043/70

Issued by U.S. Army Aviation and Missile Command Under

DAAH01-03-C-R236

Table of Contents

The SableCC (version 2.16.2) Grammar of the Dataflow Interchange Format

```
Package mapss.dif.language.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+
      (not_star_slash not_star* '*'+)* '/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';

  graph = 'graph';
  attribute = 'attribute';
  basedon = 'basedon';
  interface = 'interface';
  parameter = 'parameter';
  refinement = 'refinement';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
  dot_identifier = non_digit (digit | non_digit)*
      ('.' non_digit (digit | non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  graph_list = graph_block*;
  graph_block = identifier name l_bkt block* r_bkt;
  block =
    {basedon}                    basedon basedon_body |
    {topology}                   topology topology_body |
    {interface}                  interface interface_body |
    {parameter}                  parameter parameter_body |
    {refinement}                 refinement refinement_body |
    {builtin_attribute}          identifier attribute_body |
    {user_defined_attribute}     attribute name attribute_body |
    {actor}                      actor name actor_body;

  name = {identifier} identifier |
         {string_identifier} string_identifier;

  /***********************************
   * Definitions for basedon block:
   */

  basedon_body = l_bkt basedon_expression r_bkt;
  basedon_expression = name semicolon;

  /***********************************
```

```
 * Definitions for topology block:
 */

topology_body = l_bkt topology_list* r_bkt;
topology_list =
  {nodes}  nodes equal name node_identifier_tail* semicolon |
  {edges}  edges equal edge_definition edge_definition_tail*
           semicolon ;

node_identifier_tail = comma name;
edge_definition = [edge]:name l_par
                  [source]:name comma
                  [sink]:name r_par;
edge_definition_tail = comma edge_definition;

/*************************************
 * Definitions for interface block:
 */

interface_body = l_bkt interface_expression* r_bkt;
interface_expression =
  {input}   inputs equal port_definition port_definition_tail*
            semicolon |
  {output}  outputs equal port_definition port_definition_tail*
            semicolon;

port_definition = {plain} name |
                  {node}  [port]:name colon [node]:name;
port_definition_tail = comma port_definition;

/*************************************
 * Definitions for parameter block:
 */

parameter_body = l_bkt parameter_expression* r_bkt;
parameter_expression =
  {value}     name equal value semicolon |
  {range}     name colon range_block semicolon |
  {blank}     name semicolon;

range_block = range range_tail*;
range =
  {closed_closed}    l_sqr [left]:number comma [right]:number r_sqr |
  {open_closed}      l_par [left]:number comma [right]:number r_sqr |
  {closed_open}      l_sqr [left]:number comma [right]:number r_par |
  {open_open}        l_par [left]:number comma [right]:number r_par |
  {discrete}         l_bkt number discrete_range_number_tail* r_bkt;
discrete_range_number_tail = comma number;
range_tail = plus range;
number = {double} double | {integer} integer;

/*************************************
 * Definitions for refinement block:
 */

refinement_body = l_bkt refinement_definition refinement_expression*
                  r_bkt;
```

```
refinement_definition = [graph]:name equal [node]:name semicolon;
refinement_expression =
  {ports}     [port]:name colon [element]:name semicolon |
  {params}    [subparam]:name equal [param]:name semicolon;

/*************************************
 * Definitions for attribute block:
 */

attribute_body = l_bkt attribute_expression* r_bkt;
attribute_expression =
  {value} name? equal value semicolon |
  {reference} [element]:name? equal [reference]:name semicolon |
  {subelement_assign} [trggraph]:name [fst]:dot [trgele]:name equal
                      [srcgraph]:name [snd]:dot [srcele]:name
                      semicolon |
  {idlist} name? equal id_list semicolon;

id_list = name ref_id_tail+;
ref_id_tail = comma name;

/*************************************
 * Definitions for actor block:
 */

actor_body = l_bkt actor_expression* r_bkt;
actor_expression =
  {value} name type? equal value semicolon |
  {reference} [argument]:name type? equal [reference]:name
              semicolon |
  {reflist} name type? equal id_list semicolon;

type =
  {identifier} colon identifier |
  {dot_identifier} colon dot_identifier;

/*************************************
 * Definitions for value:
 */

value =
  {integer} integer |
  {double} double |
  {complex} l_par [real]:double comma [imag]:double r_par |
  {int_matrix} l_sqr int_row int_row_tail* r_sqr |
  {double_matrix} l_sqr double_row double_row_tail* r_sqr |
  {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
  {string} concatenated_string_value |
  {boolean} boolean_value |
  {array} l_bkt value value_tail* r_bkt;

int_row = integer integer_tail*;
integer_tail = comma integer;
int_row_tail = semicolon int_row;

double_row = double double_tail*;
double_tail = comma double;
```

```
double_row_tail = semicolon double_row;

complex = l_par [real]:double comma [imag]:double r_par;
complex_row = complex complex_tail*;
complex_tail = comma complex;
complex_row_tail = semicolon complex_row;

concatenated_string_value = string string_tail*;

boolean_value =
  {true} true |
  {false} false;

value_tail = comma value;
```

The SableCC (version 2.16.2) Grammar of the Actor Interchange Format

```
Package mapss.dif.aif.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)*
'/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';
  map_to = '->';
  map_from = '<-';
  given_that = '|';

  graph = 'graph';
  interface = 'interface';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
  dot_identifier = non_digit (digit | non_digit)*
                   ('.' non_digit (digit | non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  aif_list = aif_block*;
  aif_block = {actor} actor [trg]:type map_from [src]:type
                      method_expression? l_bkt attribute_body* r_bkt |
              {graph} graph [trg]:type map_from [src]:type
                      method_expression? l_bkt block* r_bkt;

  type = {identifier} identifier |
         {dot_identifier} dot_identifier;
  method_expression = given_that identifier
                      l_par argument argument_tail* r_par;
  argument = {id} identifier |
             {value} value;
  argument_tail = comma argument;

  /*************************************
   * Definitions for attribute body:
   */

  attribute_body = {mapping} attribute_mapping |
                   {assign} attribute_assign;
  attribute_assign = attribute equal value semicolon;
  attribute_mapping = {single} [trg]:attribute map_from [src]:attribute
                               method_expression? semicolon |
                      {multi_to_one} attribute map_from attributes
semicolon |
                      {one_to_multi} attributes map_from attribute
semicolon;

  attributes = attribute attribute_tail+;
  type_expression = colon type;
  attributes = attribute attribute_tail+;
  attribute_tail = comma attribute;

  /*************************************
   * Definitions for block:
   */

  block = {topology} topology topology_body |
          {interface} interface interface_body |
          {actor} actor name actor_body;

  name = {identifier} identifier |
         {string_identifier} string_identifier;

  /*************************************
   * Definitions for topology block:
   */

  topology_body = l_bkt topology_list* r_bkt;
  topology_list =
    {nodes}  nodes equal name node_identifier_tail* semicolon |
    {edges}  edges equal edge_definition edge_definition_tail*
semicolon ;

  node_identifier_tail = comma name;
  edge_definition = [edge]:name l_par
                    [source]:name comma
                    [sink]:name r_par;
  edge_definition_tail = comma edge_definition;

  /*************************************
   * Definitions for interface block:
   */

  interface_body = l_bkt interface_expression* r_bkt;
  interface_expression =
    {input}   inputs equal port_definition port_definition_tail*
semicolon |
    {output}  outputs equal port_definition port_definition_tail*
semicolon;

  port_definition = {plain} name port_mapping? |
                    {node}  [port]:name colon [node]:name
port_mapping?;
  port_definition_tail = comma port_definition;
  port_mapping = map_from attribute;

  /*************************************
   * Definitions for actor block:
   */
  actor_body = l_bkt actor_expression* r_bkt;
  actor_expression =
    {value} name type_expression? equal value semicolon |
    {reference} [argument]:name type_expression? equal
                [reference]:name semicolon |
    {map} name type_expression? map_from attribute
          method_expression? semicolon |
    {multi_map} name type_expression? map_from attributes semicolon |
    {reflist} name type_expression? equal id_list semicolon;

  id_list = name ref_id_tail+;
  ref_id_tail = comma name;

/**************************************
   * Definitions for value:
   */

  value =
    {integer} integer |
    {double} double |
    {complex} l_par [real]:double comma [imag]:double r_par |
    {int_matrix} l_sqr int_row int_row_tail* r_sqr |
    {double_matrix} l_sqr double_row double_row_tail* r_sqr |
    {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
    {string} concatenated_string_value |
    {boolean} boolean_value |
    {array} l_bkt value value_tail* r_bkt;

  int_row = integer integer_tail*;
  integer_tail = comma integer;
  int_row_tail = semicolon int_row;

  double_row = double double_tail*;
  double_tail = comma double;
  double_row_tail = semicolon double_row;

  complex = l_par [real]:double comma [imag]:double r_par;
  complex_row = complex complex_tail*;
  complex_tail = comma complex;
  complex_row_tail = semicolon complex_row;

  concatenated_string_value = string string_tail*;

  boolean_value =
    {true} true |
    {false} false;

  value_tail = comma value;
```

Appendix D to Final Report to

DIF - A Language for Dataflow Graph Specification and Exchange

October 27, 2004

Sponsored by

Defense Advanced Research Projects Agency (DOD)
(Controlling DARPA Office)

ARPA Order C043/70

Issued by U.S. Army Aviation and Missile Command Under

DAAH01-03-C-R236

Table of Contents

```
graph ptolemy.actor.TypedCompositeActor <- D_FFT
        | ifExpression("FI == 1 && M != N") {
    topology {
        nodes = IFFT, Scale, SequenceToArray, ArrayExtract,
ArrayToSequence;
        edges = e1 (IFFT, Scale),
                e2 (Scale, SequenceToArray),
                e3 (SequenceToArray, ArrayExtract),
                e4 (ArrayExtract, ArrayToSequence);
    }
    interface {
        inputs = in : IFFT <- X;
        outputs = out : ArrayToSequence <- Y;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order : PARAMETER <- N | conditionalAssign(
            "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) ==
0");
        input : INPUT = in;
        output : OUTPUT = e1;
    }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        input : INPUT = e1;
        output : OUTPUT = e2;
        factor : PARAMETER <- N;
    }
    actor SequenceToArray {
        computation = "ptolemy.domains.sdf.lib.SequenceToArray";
        input : INPUT = e2;
        output : OUTPUT = e3;
        arrayLength : PARAMETER <- N;
    }
    actor ArrayExtract {
        computation = "ptolemy.actor.lib.ArrayExtract";
        input : INPUT = e3;
        output : OUTPUT = e4;
        sourcePosition : PARAMETER <- B | assign("B-1");
        extractLength : PARAMETER <- M;
        destinationPosition : PARAMETER = 0;
        outputArrayLength : PARAMETER <- M;
    }
    actor ArrayToSequence {
        computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
        input : INPUT = e4;
        output : OUTPUT = out;
        arrayLength : PARAMETER <- M;
    }
}

actor ptolemy.domains.sdf.lib.FFT <- D_FFT | ifExpression("FI == 0") {
    order : PARAMETER <- N | conditionalAssign(
        "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.domains.sdf.lib.IFFT <- D_FFT | ifExpression("FI == 1") {
    order : PARAMETER <- N | conditionalAssign(
        "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT <- X;
    output : OUTPUT <- Y;
}


actor mapss.applications.sar.MatrixTranspose <- D_MTRAN {
    rowN : PARAMETER <- M;
    colN : PARAMETER <- N;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.lib.MultiplyDivide <- D_VMUL {
    multiply : INPUT <- X, Y;
    output : OUTPUT <- Z;
}

actor mapss.applications.sar.SequencePad <- D_VFILL {
    inputLength : PARAMETER <- N;
    outputLength : PARAMETER <- P | assign("P+N");
    padValue : PARAMETER <- V;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.TypedCompositeActor <- SUBGRAPH {}
```